

AX Portal Software Function Reference

Original Instructions

© Seiko Epson Corporation 2026

Rev.2
ENM268S9127M

Table of Contents

- Python Basics 3
- Core Functions 12
- Add-on Functions 160
 - Cognex Add-on Functions 161
 - Modbus Add-on Functions 167
 - REST Add-on Functions 171
 - ROS Add-on Functions 177

Python Basics

Basic Python Standards

This section explains some of the basic [Python](#) programming that can be used in the application editor. This is merely a summary of the most important Python (3.13) standards, of course there are many more Python built-in functions available. Please keep in mind that a robot application is not just a Python application, hence some Python functionalities are restricted or cannot be used. For advanced Python users or programming tutorials, we hereby refer to the main online [Python 3.13 Documentation](#).

Commentary

The following code lines represent code comments which are not executed on run-time.

```
# this is a commentary and will not be considered as executable code
"""
This is a multiline commentary
that will also not be executed.
"""
```

Variable Definitions

Python supports many different variable types and does not differentiate between variable definition and variable declaration.

```
my_boolean = True           # defining "my_boolean" as a Boolean with value True
my_integer = 3              # defining "my_integer" as an integer number with value 3
my_float = 3.1416           # defining "my_float" as a floating-point number 3.1416
my_string = "Hello World"   # defining "my_string" as a string with value "Hello
                             World"
```

Variable Conversions

Variables can easily be converted from one type to another, if the values are compatible.

```
x = bool(x)      # converts x to a boolean
x = int(x)       # converts x to an integer number
x = float(x)     # converts x to a floating-point number
x = str(x)       # converts x to a string
```

Data Structures

Python supports four different data structure types:

- A List contains an ordered array of elements.

```
my_list = list()           # create an empty list
my_list = [1, 2, 3, 4, 5]  # create a list containing some numbers
my_list.append(6)          # add element to existing list
my_list[0] = 7             # sets the first list element to be 7
```

- A Tuple contains an ordered but immutable (unchangeable) array of elements.

```
my_tuple = tuple()           # create an empty tuple
my_tuple = (1, 2, 3, 4, 5)   # create a tuple containing some numbers
```

- A Set contains an unordered array of elements that contains no duplicates.

```
my_set = set()               # create an empty set
my_set = {1, 2, 3, 4, 5}     # create a set containing some numbers
my_set.intersection({1, 3, 5, 7, 9}) # intersection of two sets
my_set.union({1, 3, 5, 7, 9})  # union of two sets
```

- A Dictionary contains an unordered combination of keys and values.

```
my_dictionary = dict()       # create an empty dictionary
my_dictionary = {"a": 1, "b": 2} # create a dictionary containing key-value pairs
my_dictionary["a"] = 3        # sets the element "a" to be 3
```

Libraries and Imports

Python provides a set of built-in modules (the standard library). The following modules are supported:

- Numerical operations and randomness – useful for calculations, mathematical functions, and generating random values

```
import math, random, statistics
```

- Time and date handling – working with timestamps, delays, and date/time values

```
import time, datetime
```

- Data storage and file formats – reading and writing structured data such as JSON or CSV formats

```
import json, csv
```

- Text processing – string manipulation and pattern matching using regular expressions

```
import re, string
```

- Advanced iteration and functional tools – efficient looping, combinations, and higher-order functions

```
import itertools, functools
```

In addition to the standard library, the following third-party libraries are available:

- Numerical computing – efficient handling of arrays and numerical data

```
import numpy
```

- 3D orientation math – quaternion representation and transformations (robotics-related)

```
import pyquaternion
```

- HTTP communication – sending requests to web services and REST APIs

```
import requests
```

- Configuration and data serialization – working with YAML formatted data

```
import yaml
```

The following standard Python modules are not available within robot applications:

- Operating system access (e.g. `os`, `shutil`, `subprocess`, `sys`)
- Custom parallelization (e.g. `multiprocessing`, `threading`)
- Custom inspection and troubleshooting (e.g. `inspect`, `traceback`)

In addition, installation of custom third-party libraries is not supported in the robot Python environment.

Basic Python Functions

Mandatory and Optional Function Arguments

Let's consider that a function has been defined with the following arguments:

```
def function(arg1, arg2, arg3="x", arg4="y"):
    ...
```

The function arguments `arg1` and `arg2` are standard arguments that need to be given when calling this function. The arguments `arg3` and `arg4` have a default value and therefore do not necessarily be given when calling this function. All the following function calls are valid:

```
function("a", "b")           # -> function("a", "b", "x", "y")
function("a", "b", "c")      # -> function("a", "b", "c", "y")
function("a", "b", "c", "d")  # -> function("a", "b", "c", "d")
function("a", "b", arg4="d")   # -> function("a", "b", "x", "d")
function(arg1="a", arg2="b", arg3="c", arg4="d") # -> function("a", "b", "c", "d")
function(arg3="c", arg2="b", arg1="a")          # -> function("a", "b", "c", "y")
```

Unpacked Function Arguments

A few functions mentioned in the code documentation use unpacked arguments. These arguments are useful if the number of arguments of a function may vary. Let's consider the following functions arguments:

```
function1(*arguments)    # function with unpacked list argument
function2(**arguments)    # function with unpacked dictionary argument
```

Then these functions can be used in the following way:

```
function1(value1, value2, ...)    # function with unpacked list argument
function2(arg1=value1, arg2=value2, ...) # function with unpacked dict argument
```

Basic Python Operators

Calculation Operators

```
x = 3 + 5    # addition:      the value of x is now 8
x = 7 - 2    # subtraction:   the value of x is now 5
x = 4 * 8    # multiplication: the value of x is now 32
x = 54 / 6   # division:      the value of x is now 9
x = 16 % 7   # modulo:        the value of x is now 2
x = 2 ** 3   # exponential:   the value of x is now 8
```

Comparison Operators

Here `x` and `y` can be of any type.

```
x == y    # returns true if x is equal to y
x != y    # returns true if x is not equal to y
```

Here `x` and `y` are numbers.

```
x > y    # returns true if x is greater than y
x < y    # returns true if x is smaller than y
x >= y   # returns true if x is greater than or equal to y
x <= y   # returns true if x is smaller than or equal to y
```

Logical Operators

Here x and y are Booleans.

```
x and y  # logical AND returns True if x and y are both True
x or y   # logical OR returns True if either x or y is True
not x    # logical NOT returns True if x is False
```

Bitwise Operators

Here x and y are integers.

```
x & y    # bitwise AND returns an integer resulting from bitwise Logical AND of x and y
x | y    # bitwise OR returns an integer resulting from bitwise Logical OR of x and y
```

Conditional Programming

If-Else Condition

`if`, `elif` (else if) and `else` conditions can be combined according to the following rules:

- The `if` scope is executed if and only if its condition returns the value True.
- An `elif` (else if) scope is optional and executed if and only if all previous `if` or `elif` conditions return the value False and its own condition returns the value True.
- An `else` scope is optional and executed if and only if all previous `if` or `elif` conditions returned the value False.

The general if-elif-else code looks as follows:

```
if condition_1:
    # the following code is executed if 'condition_1' is True
    ...
elif condition_2:
    # the following code is executed if 'condition_1' is False
    # and 'condition_2' is True
    ...
elif condition_3:
    # the following code is executed if 'condition_1' and 'condition_2' are False
    # and 'condition_3' is True
    ...
else:
    # the following code is executed if all previous conditions leading up
    # to this 'else' statement are False
    ...
```

Examples:

Checking if a variable `my_integer` is equal to 5, 10, or neither one of those numbers.

```
if my_integer == 5:
    print("your integer is 5.")
    print("have a nice day.")
elif my_integer == 10:
    print("your integer is 10.")
    print("have a nice day.")
else:
    print("your integer is neither 5 nor 10.")
    print("have a nice day anyway.")
```

While Loop

A while loop is executed and repeated according to the following rules:

- The `while` loop scope is executed repeatedly as long as its condition returns the value `True`.
- A `continue` keyword is optional and can be used to jump to the next loop iteration ignoring the remaining code within the while loop.
- A `break` keyword is optional and can be used to jump out of this loop.
- An `else` scope is optional (and rarely used) and executed if the loop condition turns to `False`. In other words, this scope is not executed if the while loop exits with a `break` statement.

```
while condition:
    # the following code is executed repeatedly as long as 'condition' is True
    ...
    if condition_1:
        # the 'continue' keyword skips the remaining code of this loop
        # and jump to the next iteration
        continue
    if condition_2:
        # the 'break' keyword breaks out of this loop, no matter the loop condition
        break
else:
    # the following code is executed only if the while loop exits
    # by setting 'condition' to False
    ...
```

Examples:

Printing all numbers from 1 to 10.

```
index = 1
while index <= 10:
    print(index)
    index += 1
```

Infinite Loop: Infinite loops are useful when you want the robot to keep repeating a task until you manually stop it, or break out of the loop using a specific condition or function. If the loop tends to run arbitrarily fast, it is recommended to add waiting commands to give the processor some time to breathe.

```
import time
while True:
    ...
    time.sleep(0.1)
```

For Loop

A for loop is executed and repeated according to the following rules:

- The `for` loop scope is executed repeatedly, once for each item in an iterable sequence (e.g. a list, a set, a dictionary, or a string).
- A `continue` keyword is optional and can be used to jump to the next loop iteration ignoring the remaining code within the for loop.
- A `break` keyword is optional and can be used to jump out of this loop.
- An `else` scope is optional (and rarely used) and executed after the last item in the for loop sequence has been processed. In other words, this scope is not executed if the for loop exits with a brake statement.

```
for item in sequence:
    # the following code is executed once for each 'item' in 'sequence'
    ...
    if condition_1:
        # the 'continue' keyword skips the remaining code of this loop
        # and jump to the next iteration
        continue
    if condition_2:
        # the 'break' keyword breaks out of this loop, no matter the loop condition
        break
else:
    # the following code is executed only if the for loop exits
    # by finding no more 'item' in 'sequence'.
    ...
```

Examples:

Printing all numbers from 1 to 5.

```
for index in range(5):  
    print(index+1)
```

Printing my favorite fruits.

```
for fruit in ["apple", "banana", "orange"]:  
    print(f"one of my favorite fruits is {fruit}")
```

Counting out the letters in `python`.

```
index = 1  
for letter in "python":  
    print(f"letter number {index} in 'python' is {letter}")  
    index += 1
```

Core Functions

class Core

Note

This is a collection of all core functions. These functions are directly available and callable in the code of applications.

`move_joints(joints, joint_position, joint_velocity=None, joint_acceleration=None, relative=False, block=True)`

Moving one or multiple joints of the robot to a particular set of joint angles. This movement synchronizes all joints such that they reach their target position at the same time.

Parameters:

- **joints** (*int, str, list*) –
The IDs of the actuators to move. Specify the value as follows:
 - the ID of a single joint (e.g. 6)
 - a list of joint IDs for multiple joints (e.g. [1, 4, 6])
 - the constant `ALL_KIN_JOINTS` to move all six joints
- **joint_position** (*float, list of float*) –
The angular positions of the moving joints in [deg]. Specify the value as follows:
 - a single number to move all specified joints to the same position (e.g. -30)
 - a list of numbers to move each specified joint to a different position (e.g. [90, -45, 30])
- **joint_velocity** (*float or None*) – The maximum velocity in percent of the velocity limit for the joints. This is a single number to limit all joints to the same maximum value (e.g. 45 percent). If no value is specified, the global joint velocity is used.
- **joint_acceleration** (*float or None*) – The maximum acceleration in percent of the acceleration limit for the moving joints. This is a single number to limit all joints to the same maximum value (e.g. 60 percent) If no value is specified, the global joint acceleration is used.
- **relative** (*bool*) – Whether the joint positions are considered as a relative offset to the current joint positions. By default, absolute position values are used.
- **block** (*bool*) – Whether to wait with the further execution of the program until this movement terminated. Note that only synchronized motions using the same joints can be executed in succession in a non-blocking manner.

Examples:

Moving all joints of the robot synchronized to 0°, which is the standing upright position of the robot:

```
# these are equivalent  
move_joints(ALL_KIN_JOINTS, 0)  
move_joints([1, 2, 3, 4, 5, 6], [0, 0, 0, 0, 0, 0])
```

Moving joint 6 by -30° from its current position:

```
move_joints(6, -30, relative=True)
```

Moving joints 2, 3 and 5 to 20° , -40° and 20° , respectively. The motion will be performed such that each joint finishes its movement at the same time. This motion will use the maximum allowed joint acceleration, but only 50% of the maximum allowed joint velocity.

```
move_joints([2, 3, 5], [20, -40, 20], joint_velocity=50,  
            joint_acceleration=100)
```

```
move_coordinates(coordinates, configuration=None, joint_velocity=None,  
joint_acceleration=None, tool_offset=None, tool_frame=None, work_offset=None,  
work_frame=None, relative=False, block=True)
```

Moving the TCP (tool center point) of the robot to a specific set of Cartesian coordinates. The robot moves along a time optimal trajectory, which results in trapezoidal velocity profiles in each joint.

Parameters:

- **coordinates** (*Coordinates, list or dict*) –
The target coordinates of the TCP. Specify the value as follows:
 - a Coordinates object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
 - a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}
- **configuration** (*str or None*) – The configuration of the target pose. This is a string from 'c1' through 'c8'.
- **joint_velocity** (*float or None*) – The maximum velocity in percent of the velocity limit for the joints. This is a single number to limit all joints to the same maximum value (e.g. 45 percent). If no value is specified, the global joint velocity is used.
- **joint_acceleration** (*float or None*) – The maximum acceleration in percent of the acceleration limit for the moving joints. This is a single number to limit all joints to the same maximum value (e.g. 60 percent) If no value is specified, the global joint acceleration is used.
- **tool_offset** (*Coordinates, list, dict, or None*) –
The transform to shift the TCP of the robot, given relative to the tool frame. Specify the value as follows:
 - a Coordinates object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
 - a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}

By default, there is no tool offset, such that the TCP lies at the origin of the tool frame.

- **tool_frame** (*Coordinates, list, dict, or None*) –
The transform to alter the TCP of the robot, given relative to the flange frame. Specify the value as follows:
 - a Coordinates object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
 - a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}

By default, the global tool frame is applied. If the tool frame contains only zero coordinates, it coincides with the flange frame.

- **work_offset** (*Coordinates, list, dict, or None*) –
The transform to shift the reference of the robot, given relative to the work frame. Specify the value as follows:

- a Coordinates object
- a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
- a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}

By default, there is no work offset, such that the reference lies at the origin of the work frame.

- **work_frame** (*Coordinates, list, dict, or None*) –

The transform to alter the reference of the robot, given relative to the base frame. Specify the value as follows:

- a Coordinates object
- a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
- a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}

By default, the global work frame is applied. If the work frame contains only zero coordinates, it coincides with the base frame.

- **relative** (*bool*) – Whether the coordinates are considered as a relative offset to the current coordinates. By default, absolute coordinate values are used.
- **block** (*bool*) – Whether to wait with the further execution of the program until this movement terminated. By default, this function is blocking.

Examples:

The target coordinates can be given in various forms. All the specified commands move the TCP to the position x = 700mm, y = -125mm, z = 500mm and orientation angles roll = 180°, pitch = 0° and yaw = -90° in the global work frame.

```
move_coordinates([700.0, -125.0, 500.0, 180.0, 0.0, -90.0])
move_coordinates({"x": 700.0, "y": -125.0, "z": 500.0,
                  "roll": 180.0, "pitch": 0.0, "yaw": -90.0})
move_coordinates(Coordinates([700.0, -125.0, 500.0, 180.0, 0.0, -90.0]))
```

If not all coordinates are specified, the current TCP coordinates are used for the remaining coordinates. For example, only changing the x and roll-component can be done as follows:

```
move_coordinates([500.0, None, None, 150.0, None, None])
move_coordinates({"x": 500.0, "roll": 150.0})
```

In the above examples, no optional parameters were specified. Therefore, the global parameters are used. The parameters can also be explicitly defined in the function call:

```
move_coordinates({"y": -100.0, "yaw": -60.0},
                  joint_velocity=25, joint_acceleration=100,
                  relative=True, tool_offset=[20, 0, 0, 0, 0, 0])
```

which moves an additional -100mm in y and -60° in yaw direction with 50% of the maximum velocity and with the maximum acceleration. In addition, the TCP is shifted by 20mm in the x-direction by the tool offset.

There are multiple options to define movement in tool space, one of them is:

```
move_coordinates(get_reference_coordinates(), tool_offset=[0.0, 0.0, 100.0,  
0.0, 0.0, 0.0])
```

which moves along the z-direction of the tool.

```
move_pose(pose, joint_velocity=None, joint_acceleration=None, tool_offset=None,  
tool_frame=None, work_offset=None, work_frame=None, block=True)
```

Moving the TCP (tool center point) of the robot to a specific pose. The robot moves along a time optimal trajectory, which results in trapezoidal velocity profiles in each joint.

Parameters:

- **pose** (*Pose, str, or int*) –
The target pose of the TCP. Specify the value as follows:
 - a Pose object
 - the name of a predefined pose (saved in the database)
 - the ID of a predefined pose (saved in the database)
- **joint_velocity** (*float or None*) – The maximum velocity in percent of the velocity limit for the joints. This is a single number to limit all joints to the same maximum value (e.g. 45 percent). If no value is specified, the global joint velocity is used.
- **joint_acceleration** (*float or None*) – The maximum acceleration in percent of the acceleration limit for the moving joints. This is a single number to limit all joints to the same maximum value (e.g. 60 percent) If no value is specified, the global joint acceleration is used.
- **tool_offset** (*Coordinates, list, dict, or None*) –
The transform to shift the TCP of the robot, given relative to the tool frame. Specify the value as follows:
 - a Coordinates object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
 - a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}By default, there is no tool offset, such that the TCP lies at the origin of the tool frame.
- **tool_frame** (*Coordinates, list, dict, or None*) –
The transform to alter the TCP of the robot, given relative to the flange frame. Specify the value as follows:
 - a Coordinates object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
 - a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}By default, the global tool frame is applied. If the tool frame contains only zero coordinates, it coincides with the flange frame.
- **work_offset** (*Coordinates, list, dict, or None*) –
The transform to shift the reference of the robot, given relative to the work frame. Specify the value as follows:
 - a Coordinates object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
 - a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}

By default, there is no work offset, such that the reference lies at the origin of the work frame.

- **work_frame** (*Coordinates, list, dict, or None*) –
The transform to alter the reference of the robot, given relative to the base frame. Specify the value as follows:
 - a Coordinates object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
 - a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}

By default, the global work frame is applied. If the work frame contains only zero coordinates, it coincides with the base frame.

- **block** (*bool*) – Whether to wait with the further execution of the program until this movement terminated. By default, this function is blocking.

Examples:

Moving to a pose from the database:

```
move_pose("fancy_pose")
move_pose(10)  # ID of the pose in the database
```

A custom pose can also be specified in the script using the `create_pose()` function:

```
pose = create_pose([700.0, -125.0, 500.0, 180.0, 0.0, -90.0], "c8")
move_pose(pose)
```

In these examples, no optional parameters were specified. Therefore, the global parameters are used. The parameters can also be explicitly defined in the function call:

```
move_pose("final_pose", joint_velocity=45, joint_acceleration=60,
          work_offset=[0, 0, 50, 0, 0, 0], block=False)
```

which moves the robot to the "final_pose" of the database with 45% of the maximum joint velocity and 60% of the maximum joint acceleration. In addition, an offset is applied to the work frame shifting the target pose by 50mm in the z-direction.

There are multiple options to define movement in tool space, one of them is:

```
move_pose(get_reference_pose(), tool_offset=[0.0, 0.0, 100.0, 0.0, 0.0, 0.0])
```

which moves along the z-direction of the tool.

```
move_linear(coordinates, linear_velocity=None, linear_acceleration=None,  
tool_offset=None, tool_frame=None, work_offset=None, work_frame=None,  
relative=False, block=True)
```

Moving the TCP (tool center point) of the robot from its current pose to a target pose in a straight line. Note that both the position and the orientation are linearly interpolated along the trajectory. Note that this movement can only be executed if each pose along the path is reachable.

Parameters:

- **coordinates** (*Coordinates, list, or dict*) –
The target coordinates of the TCP. Specify the value as follows:
 - a Coordinates object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
 - a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}
- **linear_velocity** (*float or None*) – The maximum linear tool velocity [mm/s] for all moving parts of the robot.
- **linear_acceleration** (*float or None*) – The maximum linear tool acceleration [mm/s²] for all moving parts of the robot.
- **tool_offset** (*Coordinates, list, dict, or None*) –
The transform to shift the TCP of the robot, given relative to the tool frame. Specify the value as follows:
 - a Coordinates object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
 - a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}

By default, there is no tool offset, such that the TCP lies at the origin of the tool frame.

- **tool_frame** (*Coordinates, list, dict, or None*) –
The transform to alter the TCP of the robot, given relative to the flange frame. Specify the value as follows:
 - a Coordinates object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
 - a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}

By default, the global tool frame is applied. If the tool frame contains only zero coordinates, it coincides with the flange frame.

- **work_offset** (*Coordinates, list, dict, or None*) –
The transform to shift the reference of the robot, given relative to the work frame. Specify the value as follows:
 - a Coordinates object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
 - a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}

By default, there is no work offset, such that the reference lies at the origin of the work frame.

- **work_frame** (*Coordinates, list, dict, or None*) – The transform to alter the reference of the robot, given relative to the base frame. Specify the value as follows:
 - a Coordinates object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
 - a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}

By default, the global work frame is applied. If the work frame contains only zero coordinates, it coincides with the base frame.

- **relative** (*bool*) – Whether the coordinates are considered as a relative offset to the current coordinates. By default, absolute coordinate values are used.
- **block** (*bool*) – Whether to wait with the further execution of the program until this movement terminated. By default, this function is blocking.

Examples:

The target coordinates can be given in various forms. All the specified commands move the TCP linearly to the position x = 600mm, y = -125mm, z = 500mm and orientation angles roll = 180°, pitch = 0° and yaw = -90° in the global work frame.

```
move_linear([600.0, -125.0, 500.0, 180.0, 0.0, -90.0])
move_linear({"x": 600.0, "y": -125.0, "z": 500.0,
            "roll": 180.0, "pitch": 0.0, "yaw": -90.0})
move_linear(Coordinates([600.0, -125.0, 500.0, 180.0, 0.0, -90.0]))
```

If not all coordinates are specified, the current TCP coordinates are used for the remaining coordinates. For example, the x and roll-component can be changed as follows:

```
move_linear([500.0, None, None, 150.0, None, None])
move_linear({"x": 500.0, "roll": 150.0})
```

In these examples, no optional parameters were specified. Therefore, the global parameters are used. The parameters can also be explicitly defined in the function call:

```
move_linear({"x": 100.0}, linear_velocity=500, linear_acceleration=1000,
            relative=True, work_frame=[20, 0, 0, 0, 0, 0])
```

which moves an additional 100mm in x direction with a velocity of 500mm/s and an acceleration of 1000mm/s². In addition, the work frame is overwritten with the given coordinates.

There are multiple options to define movement in tool space, one of them is:

```
move_linear(get_reference_coordinates(), tool_offset=[0.0, 0.0, 100.0, 0.0,  
0.0, 0.0])
```

which linearly moves along the z-direction of the tool.

```
move_circular(intermediate_position, coordinates, linear_velocity=None,
linear_acceleration=None, tool_offset=None, tool_frame=None, work_offset=None,
work_frame=None, relative=False, block=True)
```

Moving the TCP (tool center point) of the robot from its current pose to a target pose along a circular trajectory, passing through an intermediate position. Note that the orientation is linearly interpolated along the trajectory. Note that this movement can only be executed if each pose along the path is reachable.

Parameters:

- **intermediate_position** (*list*) – An intermediate position on the path towards the target coordinates. The intermediate position determines the curvature of the circular motion.
- **coordinates** (*Coordinates, list, or dict*) –
The target coordinates of the TCP. Specify the value as follows:
 - a Coordinates object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
 - a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}
- **linear_velocity** (*float or None*) – The maximum linear tool velocity [mm/s] for all moving parts of the robot.
- **linear_acceleration** (*float*) – The maximum linear tool acceleration [mm/s²] for all moving parts of the robot.
- **tool_offset** (*Coordinates, list, dict, or None*) –
The transform to shift the TCP of the robot, given relative to the tool frame. Specify the value as follows:
 - a Coordinates object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
 - a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}

By default, there is no tool offset, such that the TCP lies at the origin of the tool frame.

- **tool_frame** (*Coordinates, list, dict, or None*) –
The transform to alter the TCP of the robot, given relative to the flange frame. Specify the value as follows:
 - a Coordinates object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
 - a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}

By default, the global tool frame is applied. If the tool frame contains only zero coordinates, it coincides with the flange frame.

- **work_offset** (*Coordinates, list, dict, or None*) –
The transform to shift the reference of the robot, given relative to the work frame. Specify the value as follows:
 - a Coordinates object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]

- a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}

By default, there is no work offset, such that the reference lies at the origin of the work frame.

- **work_frame** (*Coordinates, list, dict, or None*) – The transform to alter the reference of the robot, given relative to the base frame. Specify the value as follows:
 - a Coordinates object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
 - a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}

By default, the global work frame is applied. If the work frame contains only zero coordinates, it coincides with the base frame.
- **relative** (*bool*) – Whether the coordinates are considered as a relative offset to the current coordinates. By default, absolute coordinate values are used.
- **block** (*bool*) – Whether to wait with the further execution of the program until this movement terminated. By default, this function is blocking.

Examples:

The robot can be moved circularly to the target position x = 0mm, y = -500mm, z = 500mm and orientation angles roll = 180°, pitch = 0° and yaw = -90° in the global work frame. During its motion it will go through the intermediate position of x = 0mm, y = -500mm, z = 500mm.

```
move_circular([0.0, -500.0, 500.0], [-500.0, -125.0, 500.0, 180.0, 0.0, -90.0])
```

If not all coordinates are specified, the current TCP coordinates are used for the remaining coordinates.

```
move_circular([0.0, -500.0, None], [-500.0, -125.0, None, None, None, None])
move_circular([0.0, -500.0, None], {"x": -500.0, "y": -125.0})
```

In these examples, no optional parameters were specified. Therefore, the global parameters are used. The parameters can also be explicitly defined in the function call:

```
move_circular([500.0, -300.0, 100.0], [-1000.0, 0.0, 0.0, 20.0, -10.0, 0.0],
              linear_velocity=500, linear_acceleration=1000,
              tool_offset={"x": 20}, relative=True)
```

which moves the TCP to the specified coordinates with a velocity of 500mm/s and an acceleration of 1000mm/s². In addition, the TCP is shifted by 20mm in the x-direction by the tool offset.

```
move_orientation(tool_orientation, angular_velocity=None,
angular_acceleration=None, tool_offset=None, tool_frame=None, work_offset=None,
work_frame=None, relative=False, block=True)
```

Moving the TCP (tool center point) such that only the orientation changes. The robot then linearly rotates according to the desired tool orientation while keeping its position constant. Note that this movement can only be executed if each pose along the path is reachable.

Parameters:

- **tool_orientation** (*list*) – desired orientation of the tool as nautical angles (roll, pitch, yaw) [deg]
- **angular_velocity** (*float or None*) – maximum angular tool velocity [deg/s] of the TCP.
- **angular_acceleration** (*float or None*) – maximum angular tool acceleration [deg/s²] of the TCP.
- **tool_offset** (*Coordinates, list, dict, or None*) –
The transform to shift the TCP of the robot, given relative to the tool frame. Specify the value as follows:
 - a Coordinates object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
 - a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}

By default, there is no tool offset, such that the TCP lies at the origin of the tool frame.

- **tool_frame** (*Coordinates, list, dict, or None*) –
The transform to alter the TCP of the robot, given relative to the flange frame. Specify the value as follows:
 - a Coordinates object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
 - a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}

By default, the global tool frame is applied. If the tool frame contains only zero coordinates, it coincides with the flange frame.

- **work_offset** (*Coordinates, list, dict, or None*) –
The transform to shift the reference of the robot, given relative to the work frame. Specify the value as follows:
 - a Coordinates object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
 - a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}

By default, there is no work offset, such that the reference lies at the origin of the work frame.

- **work_frame** (*Coordinates, list, dict, or None*) –
The transform to alter the reference of the robot, given relative to the base frame. Specify the value as follows:
 - a Coordinates object

- a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
- a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}

By default, the global work frame is applied. If the work frame contains only zero coordinates, it coincides with the base frame.

- **relative** (*bool*) – Whether the coordinates are considered as a relative offset to the current coordinates. By default, absolute coordinate values are used.
- **block** (*bool*) – Whether to wait with the further execution of the program until this movement terminated. By default, this function is blocking.

Examples:

The target orientation can either specify all angles or only some of them and have 'None' for the others. In the latter case, the current orientation is used for the missing angles.

```
move_orientation([150.0, -20.0, -60.0])
move_orientation([150.0, None, -60.0])
```

In these examples, no optional parameters were specified. Therefore, the global parameters are used. The parameters can also be explicitly defined in the function call:

```
move_orientation([-60, 0, 0], angular_velocity=250, angular_acceleration=3000,
                 tool_frame=[0, 0, 0, 0, 20, 0], relative=True)
```

which corresponds to a relative movement of -60° in roll direction with a velocity of $250^\circ/\text{s}$ and an acceleration of $3000^\circ/\text{s}^2$. In addition, the tool frame is overwritten with the given coordinates.

There are multiple options to define movement in tool space, one of them is:

```
move_orientation(get_reference_coordinates().orientation, tool_offset=[0.0,
0.0, 0.0, 10.0, 0.0, 0.0])
```

which rotates around the z-direction of the tool.

```
move_waypoints(poses, joint_velocity=None, joint_acceleration=None,  
blend_radius=None, tool_offset=None, tool_frame=None, work_offset=None,  
work_frame=None, block=True)
```

Moves the TCP (tool center point) through the defined waypoints, starting at the current robot pose.

Parameters:

- **poses** (*list of str, int or Pose*) – A list of waypoints.
- **joint_velocity** (*float or None*) – The maximum velocity in percent of the velocity limit for the joints. This is a single number to limit all joints to the same maximum value (e.g. 45 percent). If no value is specified, the global joint velocity is used.
- **joint_acceleration** (*float or None*) – The maximum acceleration in percent of the acceleration limit for the moving joints. This is a single number to limit all joints to the same maximum value (e.g. 60 percent) If no value is specified, the global joint acceleration is used.
- **blend_radius** (*float or None*) – Allowed blend radius [mm] defining the maximum deviation from the waypoints in tool space during transitioning between waypoints.
- **tool_offset** (*Coordinates, list, dict, or None*) –
The transform to shift the TCP of the robot, given relative to the tool frame. Specify the value as follows:
 - a Coordinates object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
 - a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}

By default, there is no tool offset, such that the TCP lies at the origin of the tool frame.

- **tool_frame** (*Coordinates, list, dict, or None*) –
The transform to alter the TCP of the robot, given relative to the flange frame. Specify the value as follows:
 - a Coordinates object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
 - a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}

By default, the global tool frame is applied. If the tool frame contains only zero coordinates, it coincides with the flange frame.

- **work_offset** (*Coordinates, list, dict, or None*) –
The transform to shift the reference of the robot, given relative to the work frame. Specify the value as follows:
 - a Coordinates object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
 - a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}

By default, there is no work offset, such that the reference lies at the origin of the work frame.

- **work_frame** (*Coordinates, list, dict, or None*) –
The transform to alter the reference of the robot, given relative to the base frame. Specify the value as follows:
 - a Coordinates object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
 - a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}

By default, the global work frame is applied. If the work frame contains only zero coordinates, it coincides with the base frame.

- **block** (*bool*) – Whether to wait with the further execution of the program until this movement terminated. Note that only synchronized motions using the same joints can be executed in succession in a non-blocking manner.

Examples:

The waypoints can be given in different forms (`Pose` object, name of pose in the database, ID of pose in the database):

```
first_waypoint = create_pose([700.0, -125.0, 500.0, 180.0, 0.0, -90.0], "c8")
move_waypoints([first_waypoint, "pick_pose", 6])
```

In the above examples, no optional parameters were specified. Therefore, the global parameters are used. The parameters can also be explicitly defined in the function call:

```
move_waypoints([first_waypoint, "pick_pose", 6], joint_velocity=25,
               joint_acceleration=100, blend_radius=50)
```

which moves along the waypoints allowing a maximum deviation from them of 50mm. It uses 50% of the maximum velocity and the maximum acceleration.

```
move_segments(segments, tool_offset=None, tool_frame=None, work_offset=None,  
work_frame=None, block=True)
```

Moves the TCP (tool center point) according to the defined segments. It starts at the current robot pose.

Parameters:

- **segments** (*list*) – A list of segments that the path is made of.
- **tool_offset** (*Coordinates, list, dict, or None*) –
The transform to shift the TCP of the robot, given relative to the tool frame.
Specify the value as follows:
 - a Coordinates object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
 - a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}

By default, there is no tool offset, such that the TCP lies at the origin of the tool frame.
- **tool_frame** (*Coordinates, list, dict, or None*) –
The transform to alter the TCP of the robot, given relative to the flange frame.
Specify the value as follows:
 - a Coordinates object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
 - a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}

By default, the global tool frame is applied. If the tool frame contains only zero coordinates, it coincides with the flange frame.
- **work_offset** (*Coordinates, list, dict, or None*) –
The transform to shift the reference of the robot, given relative to the work frame. Specify the value as follows:
 - a Coordinates object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
 - a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}

By default, there is no work offset, such that the reference lies at the origin of the work frame.
- **work_frame** (*Coordinates, list, dict, or None*) –
The transform to alter the reference of the robot, given relative to the base frame. Specify the value as follows:
 - a Coordinates object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
 - a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}

By default, the global work frame is applied. If the work frame contains only zero coordinates, it coincides with the base frame.

- **block** (*bool*) – Whether to wait with the further execution of the program until this movement terminated. Note that only synchronized motions using the same joints can be executed in succession in a non-blocking manner.

Examples:

The segments can be defined in the script using the Segment objects (`LinearSegment`, `CircularSegment`, ...):

```
lin_seg = LinearSegment([700.0, -125.0, 500.0, 180.0, 0.0, -90.0])
pose_seg = PoseSegment("place_2")

move_segments([lin_seg, pose_seg])
```

In these examples, no optional parameters were specified. Therefore, the global parameters are used. The parameters can also be explicitly defined in the function call:

```
lin_seg = LinearSegment([700.0, -125.0, 500.0, 180.0, 0.0, -90.0])
pose_seg = PoseSegment("place_2")

move_segments([lin_seg, pose_seg], tool_offset=[20, 0, 0, 0, 0, 0],
              block=False)
```

which executes the segments using a slight shift in the TCP defined by the tool offset. Additionally, the movement is set to be non-blocking.

```
run_path(path, previous_angles=None, tool_offset=None, tool_frame=None,  
work_offset=None, work_frame=None, block=True)
```

Moves the TCP (tool center point) according to the specified path. First, the robot is moved to the initial pose of the path using the default velocities and accelerations. From there the segments are executed one after the other according to their specification.

Parameters:

- **path** (*Path, str, or int*) – The path the TCP should follow.
- **previous_angles** (*list or None*) –
The angles to which the nearest solution for the start pose will be calculated.
Specify the value as follows:
 - None, to use the current robot angles
 - a list of joint angles [deg]
- **tool_offset** (*Coordinates, list, dict, or None*) –
The transform to shift the TCP of the robot, given relative to the tool frame.
Specify the value as follows:
 - a Coordinates object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
 - a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}

By default, there is no tool offset, such that the TCP lies at the origin of the tool frame.
- **tool_frame** (*Coordinates, list, dict, or None*) –
The transform to alter the TCP of the robot, given relative to the flange frame.
Specify the value as follows:
 - a Coordinates object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
 - a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}

By default, the global tool frame is applied. If the tool frame contains only zero coordinates, it coincides with the flange frame.
- **work_offset** (*Coordinates, list, dict, or None*) –
The transform to shift the reference of the robot, given relative to the work frame. Specify the value as follows:
 - a Coordinates object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
 - a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}

By default, there is no work offset, such that the reference lies at the origin of the work frame.
- **work_frame** (*Coordinates, list, dict, or None*) –
The transform to alter the reference of the robot, given relative to the base frame. Specify the value as follows:
 - a Coordinates object

- a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
- a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}

By default, the global work frame is applied. If the work frame contains only zero coordinates, it coincides with the base frame.

- **block** (*bool*) – Whether to wait with the further execution of the program until this movement terminated. Note that only synchronized motions using the same joints can be executed in succession in a non-blocking manner.

Examples:

Run a path from the database:

```
run_path("fancy_path")
run_path(10) # ID of the path in the database
```

A custom path can also be specified in the script using the `create_path()` function:

```
path = create_path(start_pose, [LinearSegment(...), ...])
run_path(path)
```

Another option is to use the `Path` object to create a path and then run that one:

```
path = Path("start_pose", [])
path.add_linear([700.0, -125.0, 500.0, 180.0, 0.0, -90.0])

run_path(path)
```

In these examples, no optional parameters were specified. Therefore, the global parameters are used. The parameters can also be explicitly defined in the function call:

```
run_path("fancy_path", tool_frame=[20, 0, 0, 0, 0, 0], block=False)
```

which executes the "fancy_path" using a slight shift in the TCP defined by the tool frame. Additionally, the movement is set to be non-blocking.

wait_for_motor(*actuator_ids=None*)

Waiting for one or multiple joint motors to stop moving.

Parameters:

actuator_ids (*None, str, int, list, set*) –

The IDs of the actuators to wait for. Specify the value as follows:

- the ID of a single joint (e.g. 6)
- a list of joint IDs for multiple joints (e.g. [1, 4, 6])
- a set of joint names for multiple joints (e.g. {"1", "4", "6"})
- the constant *ALL_KIN_JOINTS* or *None* to read from all joints

Examples:

This example shows that the time while a robot moves can be used for custom purposes.

```
# move to the 'start pose'
move_pose("start_pose", block=True)
# start moving towards the 'end pose'
move_pose("end_pose", block=False)
# the idle time can be used e.g. for calculations, observations, etc.
#custom_function()
# wait for the non-blocking motion to end
wait_for_motor()
```

stop_motion()

Stopping the motion by sending a stop signal to all connected actuators.

Examples:

This example shows that non-blocking movements can be aborted using the 'stop_motion' function. Here we want to abort the movement if a specific digital input toggles.

```
# move to the 'start pose'
move_pose("start_pose", block=True)
# start checking for the value of a digital input
initial_value = read_digital_main_input(1)
# start moving towards the 'end pose'
move_pose("target_pose", block=False)
# stop the motion as soon as the value of the digital input toggles
while is_motor_moving() and read_digital_main_input(1) == initial_value:
    wait(0.1)
stop_motion()
```

is_motor_moving(*actuator_ids=None*)

Check if one or multiple motors are moving

Parameters:

actuator_ids (*None, str, int, list, set*) –

The IDs of the actuators to check. Specify the value as follows:

- the ID of a single joint (e.g. 6)
- a list of joint IDs for multiple joints (e.g. [1, 4, 6])
- a set of joint names for multiple joints (e.g. {1, 4, 6})
- the constant *ALL_KIN_JOINTS* or *None* to read from all joints

Returns:

Whether the specified motors are moving. Can be one of the following:

- a bool, if a single motor was checked
- a list of bools, if multiple motors have been checked

Return type:

bool or list of bool

Examples:

This example shows that non-blocking movements can be aborted using the 'stop_motion' function. Here we want to abort the movement if a specific digital input toggles.

```
# move to the 'start pose'
move_pose("start_pose", block=True)
# start checking for the value of a digital input
initial_value = read_digital_main_input(1)
# start moving towards the 'end pose'
move_pose("target_pose", block=False)
# stop the motion as soon as the value of the digital input toggles
while is_motor_moving() and read_digital_main_input(1) == initial_value:
    wait(0.1)
stop_motion()
```

set_global_joint_velocity(*joint_velocity*)

Setting a new global default value for the `joint_velocity` argument used in point-to-point movement functions. The joint velocity value is given in percent of the maximum joint velocity limit.

Parameters:

joint_velocity (*float*) – The global joint velocity in [%] of the maximum joint velocity.

Examples:

Calling two consecutive point-to-point movements using the same velocity limits.

```
set_global_joint_velocity(40)
move_pose("pose_1")
move_pose("pose_2")
# ... this is similar to ...
move_pose("pose_1", joint_velocity=40)
move_pose("pose_2", joint_velocity=40)
```

set_global_joint_acceleration(*joint_acceleration*)

Setting a new global default value for the `joint_acceleration` argument used in point-to-point movement functions. The joint acceleration value is given in percent of the maximum joint acceleration limit.

Parameters:

joint_acceleration (*float*) – The global joint acceleration in [%] of the maximum joint acceleration.

Examples:

Calling two consecutive point-to-point movements using the same acceleration limits.

```
set_global_joint_acceleration(80)
move_pose("pose_1")
move_pose("pose_2")
# ... this is similar to ...
move_pose("pose_1", joint_acceleration=80)
move_pose("pose_2", joint_acceleration=80)
```

set_global_linear_velocity(*linear_velocity*)

Setting a new global default value for the `linear_velocity` argument used in tool movement and path planner functions. The linear velocity value limits the maximum linear velocity of the tool and all moving parts of the robot.

Parameters:

linear_velocity (*float*) – The global linear velocity in [mm/s].

Examples:

Calling two consecutive linear movements using the same velocity limits.

```
set_global_linear_velocity(150)
move_linear("pose_1")
move_linear("pose_2")
# ... this is similar to ...
move_linear("pose_1", linear_velocity=150)
move_linear("pose_2", linear_velocity=150)
```

set_global_linear_acceleration(*linear_acceleration*)

Setting a new global default value for the `linear_acceleration` argument used in tool movement and path planner functions. The linear acceleration value limits the maximum linear acceleration of the tool and all moving parts of the robot.

Parameters:

linear_acceleration (*float*) – The global linear acceleration in [mm/s²].

Examples:

Calling two consecutive linear movements using the same acceleration limits.

```
set_global_linear_acceleration(600)
move_linear("pose_1")
move_linear("pose_2")
# ... this is similar to ...
move_linear("pose_1", linear_acceleration=600)
move_linear("pose_2", linear_acceleration=600)
```

set_global_angular_velocity(*angular_velocity*)

Setting a new global default value for the `angular_velocity` argument used in rotational movement functions. The angular velocity value limits the rotational velocity of the tool.

Parameters:

angular_velocity (*float*) – The global angular velocity in [deg/s].

Examples:

Calling two consecutive rotational movements using the same velocity limits.

```
set_global_angular_velocity(20)
move_orientation("pose_1")
move_orientation("pose_2")
# ... this is similar to ...
move_orientation("pose_1", angular_velocity=20)
move_orientation("pose_2", angular_velocity=20)
```

set_global_angular_acceleration(*angular_acceleration*)

Setting a new global default value for the `angular_acceleration` argument used in rotational movement functions. The angular acceleration value limits the rotational acceleration of the tool.

Parameters:

angular_acceleration (*float*) – The global angular acceleration in [deg/s²].

Examples:

Calling two consecutive rotational movements using the same acceleration limits.

```
set_global_angular_acceleration(45)
move_orientation("pose_1")
move_orientation("pose_2")
# ... this is similar to ...
move_orientation("pose_1", angular_acceleration=45)
move_orientation("pose_2", angular_acceleration=45)
```

set_global_blend_radius(*blend_radius*)

Setting a new global default value for the `blend_radius` argument used in movement and path functions.

Parameters:

blend_radius (*float*) – The global blend radius in [mm].

Examples:

Calling two consecutive waypoints movements using the same blend radius.

```
set_global_blend_radius(100)
move_waypoints(["pose_1", "pose_2"])
# ... this is similar to ...
move_waypoints(["pose_1", "pose_2"], blend_radius=100)
```

set_global_work_frame(*work_frame*)

Setting a new global default value for the `work_frame` argument used in movement and kinematic functions. The work frame is the transform to alter the reference frame of the robot, given relative to the base frame.

Parameters:

work_frame (*Coordinates*, *list*, or *dict*) –

The global work frame, given relative to the base frame. Specify the value as follows:

- a *Coordinates* object
- a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
- a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}

Examples:

Calling two consecutive movements using the same work frame.

```
my_work_frame = Coordinates([450, -300, 0, 0, 0, 90])

set_global_work_frame(my_work_frame)
move_pose("pose_1")
move_linear("pose_2")
# ... this is similar to ...
move_pose("pose_1", work_frame=my_work_frame)
move_linear("pose_2", work_frame=my_work_frame)
```

set_global_tool_frame(*tool_frame*)

Setting a new global default value for the `tool_frame` which is used in movement and kinematic functions. The tool frame is the transform to alter the TCP of the robot, given relative to the flange frame.

Parameters:

tool_frame (*Coordinates, list, or dict*) –

The global tool frame, given relative to the flange frame. Specify the value as follows:

- a Coordinates object
- a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
- a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}

Examples:

Calling two consecutive movements using the same tool frame.

```
my_tool_frame = Coordinates([0, 0, 65, 0, 0, 180])

set_global_tool_frame(my_tool_frame)
move_pose("pose_1")
move_linear("pose_2")
# ... this is similar to ...
move_pose("pose_1", tool_frame=my_tool_frame)
move_linear("pose_2", tool_frame=my_tool_frame)
```

```
get_current_pose(tool_offset=None, tool_frame=None, work_offset=None,  
work_frame=None)
```

Reading the current TCP (tool center point) pose of the robot. If frames are not given, the pose is read with respect to the global frames.

Parameters:

- **tool_offset** (*Coordinates, list, dict, or None*) –
The transform to shift the TCP of the robot, given relative to the tool frame.
Specify the value as follows:
 - a Coordinates object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
 - a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}

By default, there is no tool offset, such that the TCP lies at the origin of the tool frame.
- **tool_frame** (*Coordinates, list, dict, or None*) –
The transform to alter the TCP of the robot, given relative to the flange frame.
Specify the value as follows:
 - a Coordinates object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
 - a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}

By default, the global tool frame is applied. If the tool frame contains only zero coordinates, it coincides with the flange frame.
- **work_offset** (*Coordinates, list, dict, or None*) –
The transform to shift the reference of the robot, given relative to the work frame. Specify the value as follows:
 - a Coordinates object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
 - a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}

By default, there is no work offset, such that the reference lies at the origin of the work frame.
- **work_frame** (*Coordinates, list, dict, or None*) –
The transform to alter the reference of the robot, given relative to the base frame. Specify the value as follows:
 - a Coordinates object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
 - a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}

By default, the global work frame is applied. If the work frame contains only zero coordinates, it coincides with the base frame.

Returns:

The current TCP pose of the robot.

Return type:

Pose

Examples:

Reading the properties of a pose in two different ways.

```
pose = get_current_pose()
coordinates = pose.coordinates
configuration = pose.configuration
# ... this is similar to ...
coordinates = get_current_coordinates()
configuration = get_current_configuration()
```

Note that the current pose is read from sensor data. Sensors may fluctuate and not always return precise values, hence the following example of moving back and forth relative to the current pose will lead to an unexpected drift over time. In this case it's better to use the function *get_reference_pose* instead.

```
while True:
    move_pose(get_current_pose(), tool_offset=[0.0, 0.0, 100.0, 0.0, 0.0,
0.0])
    move_pose(get_current_pose(), tool_offset=[0.0, 0.0, -100.0, 0.0, 0.0,
0.0])
```

```
get_current_coordinates(tool_offset=None, tool_frame=None, work_offset=None,  
work_frame=None)
```

Reading the current coordinates of the TCP (tool center point) of the robot with respect to the given coordinate frames. If no frames are specified, it returns the coordinates with respect to the global frames.

Parameters:

- **tool_offset** (*Coordinates, list, dict, or None*) –
The transform to shift the TCP of the robot, given relative to the tool frame. Specify the value as follows:
 - a Coordinates object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
 - a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}By default, there is no tool offset, such that the TCP lies at the origin of the tool frame.
- **tool_frame** (*Coordinates, list, dict, or None*) –
The transform to alter the TCP of the robot, given relative to the flange frame. Specify the value as follows:
 - a Coordinates object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
 - a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}By default, the global tool frame is applied. If the tool frame contains only zero coordinates, it coincides with the flange frame.
- **work_offset** (*Coordinates, list, dict, or None*) –
The transform to shift the reference of the robot, given relative to the work frame. Specify the value as follows:
 - a Coordinates object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
 - a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}By default, there is no work offset, such that the reference lies at the origin of the work frame.
- **work_frame** (*Coordinates, list, dict, or None*) –
The transform to alter the reference of the robot, given relative to the base frame. Specify the value as follows:
 - a Coordinates object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
 - a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}By default, the global work frame is applied. If the work frame contains only zero coordinates, it coincides with the base frame.

Returns:

The current TCP coordinates of the robot.

Return type:

[Coordinates](#)

Examples:

Reading the current TCP coordinates in the global coordinate frames.

```
coordinates = get_current_coordinates()
```

Note that the current coordinates are read from sensor data. Sensors may fluctuate and not always return precise values, hence the following example of moving back and forth relative to the current pose will lead to an unexpected drift over time. In this case it's better to use the function *get_reference_coordinates* instead.

```
while True:
    move_coordinates(get_current_coordinates(), tool_offset=[0.0, 0.0, 100.0,
0.0, 0.0, 0.0])
    move_coordinates(get_current_coordinates(), tool_offset=[0.0, 0.0, -100.0,
0.0, 0.0, 0.0])
```

get_current_configuration()

Reading the name of the current robot pose configuration. Note that this configuration is independent of the current global frames.

Returns:

The current configuration (which is one of 'c1' through 'c8').

Return type:

str

Examples:

Reading the configuration of the current robot pose in two different ways.

```
configuration = get_current_configuration()
# ... this is similar to ...
pose = get_current_pose()
configuration = f"c{pose.configuration + 1}"
```

```
get_reference_pose(tool_offset=None, tool_frame=None, work_offset=None,  
work_frame=None)
```

Getting the reference TCP (tool center point) pose of the robot. If frames are not given, the pose is returned with respect to the global frames. Note that this pose is calculated from the reference angles sent to the drives.

Parameters:

- **tool_offset** (*Coordinates, list, dict, or None*) –
The transform to shift the TCP of the robot, given relative to the tool frame. Specify the value as follows:
 - a Coordinates object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
 - a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}By default, there is no tool offset, such that the TCP lies at the origin of the tool frame.
- **tool_frame** (*Coordinates, list, dict, or None*) –
The transform to alter the TCP of the robot, given relative to the flange frame. Specify the value as follows:
 - a Coordinates object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
 - a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}By default, the global tool frame is applied. If the tool frame contains only zero coordinates, it coincides with the flange frame.
- **work_offset** (*Coordinates, list, dict, or None*) –
The transform to shift the reference of the robot, given relative to the work frame. Specify the value as follows:
 - a Coordinates object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
 - a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}By default, there is no work offset, such that the reference lies at the origin of the work frame.
- **work_frame** (*Coordinates, list, dict, or None*) –
The transform to alter the reference of the robot, given relative to the base frame. Specify the value as follows:
 - a Coordinates object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
 - a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}By default, the global work frame is applied. If the work frame contains only zero coordinates, it coincides with the base frame.

Returns:

The reference TCP pose of the robot.

Return type:

Pose

Examples:

Reading the properties of a reference pose in two different ways.

```
pose = get_reference_pose()
coordinates = pose.coordinates
configuration = pose.configuration
# ... this is similar to ...
coordinates = get_reference_coordinates()
configuration = get_reference_configuration()
```

```
get_reference_coordinates(tool_offset=None, tool_frame=None, work_offset=None,  
work_frame=None)
```

Getting the reference coordinates of the TCP (tool center point) of the robot with respect to the given coordinate frames. If no frames are specified, it returns the coordinates with respect to the global frames. Note that these coordinates are calculated from the reference angles sent to the drives.

Parameters:

- **tool_offset** (*Coordinates, list, dict, or None*) –
The transform to shift the TCP of the robot, given relative to the tool frame.
Specify the value as follows:
 - a Coordinates object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
 - a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}By default, there is no tool offset, such that the TCP lies at the origin of the tool frame.
- **tool_frame** (*Coordinates, list, dict, or None*) –
The transform to alter the TCP of the robot, given relative to the flange frame.
Specify the value as follows:
 - a Coordinates object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
 - a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}By default, the global tool frame is applied. If the tool frame contains only zero coordinates, it coincides with the flange frame.
- **work_offset** (*Coordinates, list, dict, or None*) –
The transform to shift the reference of the robot, given relative to the work frame. Specify the value as follows:
 - a Coordinates object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
 - a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}By default, there is no work offset, such that the reference lies at the origin of the work frame.
- **work_frame** (*Coordinates, list, dict, or None*) –
The transform to alter the reference of the robot, given relative to the base frame. Specify the value as follows:
 - a Coordinates object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
 - a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}By default, the global work frame is applied. If the work frame contains only zero coordinates, it coincides with the base frame.

Returns:

The reference TCP coordinates of the robot.

Return type:

[Coordinates](#)

Examples:

Reading the reference TCP coordinates in the global coordinate frames.

```
coordinates = get_reference_coordinates()
```

get_reference_configuration()

Getting the name of the reference robot pose configuration. Note that this configuration is independent of the reference global frames. Note that this configuration is calculated from the reference angles sent to the drives.

Returns:

The reference configuration (which is one of 'c1' through 'c8').

Return type:

str

Examples:

Reading the configuration of the reference robot pose in two different ways.

```
configuration = get_reference_configuration()
# ... this is similar to ...
pose = get_reference_pose()
configuration = f"c{pose.configuration + 1}"
```

create_pose(*coordinates*, *configuration*)

Creates a Pose object based on the given coordinates and configuration.

Parameters:

- **coordinates** (*Coordinates*, *list*, or *dict*) –
The TCP coordinates of the pose. Specify the value as follows:
 - a Coordinates object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
 - a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}
- **configuration** (*str*) – The configuration of the pose. This is one of 'c1' through 'c8'.

Returns:

The robot pose object.

Return type:

Pose

Examples:

Creating a pose from a list of coordinate values, in a specific configuration.

```
pose = create_pose([350.0, 0.0, 550.0, 0.0, -30.0, 0.0], "c1")
```

get_pose(*pose*)

Reading the pose object from a predefined pose saved in the database.

Parameters:

pose (*Pose*, *str*, or *int*) –

The robot pose to read the coordinates from. Can be one of the following:

- a Pose object
- the name (str) of a pose saved in the database
- the ID (int) of a pose saved in the database

Returns:

The pose object.

Return type:

Pose

Examples:

Loading a pose from the database with name "grasping_pose".

```
pose = get_pose("grasping_pose")
```

create_path(*initial_pose*, *segments*)

Creates a Path object based on the given initial pose and segments.

Parameters:

- **initial_pose** (*Pose*) – The initial robot pose.
- **segments** (*list*) – A list of segments that the path consists of. Note that segments can also be added later on.

Returns:

The robot path object.

Return type:

Path

Examples:

Creating a path from an initial pose and multiple segments.

```
path = create_path(start_pose, [LinearSegment(...), ...])
```

get_path(*path*)

Reading the path object from a predefined path saved in the database.

Parameters:

path (*Path*, *str*, or *int*) –

The robot path to get. Can be one of the following:

- a Path object
- the name (str) of a path saved in the database
- the ID (int) of a path saved in the database

Returns:

The path object.

Return type:

Path

Examples:

Loading a path from the database with name "grasping_path".

```
pose = get_path("grasping_path")
```

```
parse_linear_segment(coordinates, linear_velocity=None, linear_acceleration=None,  
blend_radius=None)
```

Create a LinearSegment object.

Parameters:

- **coordinates** (*list, dict, or [Coordinates](#)*) – Target coordinates.
- **linear_velocity** (*float or None*) – Desired linear velocity [mm/s]. If no linear velocity is defined, the global linear velocity value will be used.
- **linear_acceleration** (*float or None*) – Desired linear acceleration [mm/s²]. If no linear acceleration is defined, the global linear acceleration value will be used.
- **blend_radius** (*float or None*) – Blend radius at the start of the segment [mm]. If no blend radius is defined, 0.0 will be used instead.

Constructor:

Creating a LinearSegment object by defining all parameters:

```
linear_seg = parse_linear_segment([100, 200, 300, 15, -15, 30],  
                                  linear_velocity=500,  
                                  linear_acceleration=1000,  
                                  blend_radius=100)
```

or using the default values:

```
linear_seg = parse_linear_segment([100, 200, 300, 15, -15, 30])
```

```
parse_circular_segment(intermediate_position, coordinates, linear_velocity=None,  
linear_acceleration=None, blend_radius=None)
```

Create a CircularSegment object.

Parameters:

- **intermediate_position** (*list*) – An intermediate position [x, y, z] on the path towards the target coordinates. The intermediate position determines the curvature of the circular motion.
- **coordinates** (*list, dict or [Coordinates](#)*) – Target coordinates.
- **linear_velocity** (*float or None*) – Desired linear velocity [mm/s]. If no linear velocity is defined, the global linear velocity value will be used.
- **linear_acceleration** (*float or None*) – Desired linear acceleration [mm/s²]. If no linear acceleration is defined, the global linear acceleration value will be used.
- **blend_radius** (*float or None*) – Blend radius at the start of the segment [mm]. If no blend radius is defined, 0.0 will be used instead.

Constructor:

Creating a CircularSegment object by defining all parameters:

```
circular_seg = parse_circular_segment([0, 300, 300], [100, 200, 300, 15, -15,  
30],  
                                     linear_velocity=500,  
linear_acceleration=1000,  
                                     blend_radius=100)
```

or using the default values:

```
circular_seg = parse_circular_segment([0, 300, 300], [100, 200, 300, 15, -15,  
30])
```

```
parse_orientation_segment(orientation, angular_velocity=None,  
angular_acceleration=None, blend_radius=None)
```

Create an OrientationSegment object.

Parameters:

- **orientation** (*list*) – Target orientation. Note that the position stays constant.
- **angular_velocity** (*float or None*) – Desired angular velocity [deg/s]. If no angular velocity is defined, the global angular velocity value will be used.
- **angular_acceleration** (*float or None*) – Desired angular acceleration [deg/s²]. If no angular acceleration is defined, the global angular acceleration value will be used.
- **blend_radius** (*float or None*) – Blend radius at the start of the segment [mm]. A blend radius of more than 0 allows the tool center point (TCP) to rotate before the position of this segment is reached. If no blend radius is defined, 0.0 will be used instead.

Constructor:

Creating an OrientationSegment object by defining all parameters:

```
orientation_seg = parse_orientation_segment([15, -15, 30],  
                                           angular_velocity=250,  
                                           angular_acceleration=500,  
                                           blend_radius=100)
```

or using the default values:

```
orientation_seg = parse_orientation_segment([15, -15, 30])
```

```
parse_coordinates_segment(coordinates, linear_velocity=None,  
linear_acceleration=None, blend_radius=None)
```

Create a CoordinatesSegment object.

Parameters:

- **coordinates** (*list, dict or [Coordinates](#)*) – Target coordinates.
- **linear_velocity** (*float or None*) – Desired linear velocity [mm/s]. If no linear velocity is defined, the global linear velocity value will be used.
- **linear_acceleration** (*float or None*) – Desired linear acceleration [mm/s²]. If no linear acceleration is defined, the global linear acceleration value will be used.
- **blend_radius** (*float or None*) – Blend radius at the start of the segment [mm]. If no blend radius is defined, 0.0 will be used instead.

Constructor:

Creating a CoordinatesSegment object by defining all parameters:

```
coordinates_seg = parse_coordinates_segment([100, 200, 300, 15, -15, 30],  
                                            linear_velocity=250,  
linear_acceleration=500,  
                                            blend_radius=100)
```

or using the default values:

```
coordinates_seg = parse_coordinates_segment([100, 200, 300, 15, -15, 30])
```

```
parse_pose_segment(pose, linear_velocity=None, linear_acceleration=None,  
blend_radius=None)
```

Create a PoseSegment object.

Parameters:

- **pose** (*Pose*) – Target pose.
- **linear_velocity** (*float or None*) – Desired linear velocity [mm/s]. If no linear velocity is defined, the global linear velocity value will be used.
- **linear_acceleration** (*float or None*) – Desired linear acceleration [mm/s²]. If no linear acceleration is defined, the global linear acceleration value will be used.
- **blend_radius** (*float or None*) – Blend radius at the start of the segment [mm]. If no blend radius is defined, 0.0 will be used instead.

Constructor:

Creating a PoseSegment object by defining all parameters:

```
pose_seg = parse_pose_segment(create_pose([100, 200, 300, 15, -15, 30], "c1"),  
                               linear_velocity=250, linear_acceleration=500,  
                               blend_radius=100)
```

or using the default values and pose from database:

```
pose_seg = parse_pose_segment("target_pose")
```

get_coordinates(*pose*)

Reading the coordinates of the specified pose. Note that these coordinates are independent of the current global frames.

Parameters:

pose (*Pose*, *str*, or *int*) –

The robot pose to read the coordinates from. Can be one of the following:

- a Pose object
- the name (str) of a pose saved in the database
- the ID (int) of a pose saved in the database

Returns:

The coordinates of this pose.

Return type:

Coordinates

Examples:

Reading the coordinates of the pose saved previously in the database with the name "grasping_pose".

```
coordinates = get_coordinates("grasping_pose")
```

Reading the coordinates of the pose saved previously in the database with ID 3.

```
coordinates = get_coordinates(3)
```

get_configuration(*pose*)

Reading the configuration of a specific pose.

Parameters:

pose (*Pose*, *str*, or *int*) –

The pose to read the configuration from. Specify the value as follows:

- a Pose object
- the name of a predefined pose (saved in the database)
- the ID of a predefined pose (saved in the database)

Returns:

The configuration of this pose (which is one of 'c1' through 'c8').

Return type:

str

Examples:

Reading the configuration of a pose saved with the name "grasping_pose" in the database.

```
configuration = get_configuration("grasping_pose")
```

Reading the configuration from a Pose object in two different ways.

```
configuration = get_configuration(pose)
# ... this is similar to ...
configuration = f"c{pose.configuration + 1}"
```

parse_coordinates(coordinates)

Create a Coordinates object.

Parameters:

coordinates (*list, dict, or [Coordinates](#)*) –

A specific set of coordinates, which can be one of the following:

- a list of the form [x, y, z, roll, pitch, yaw] with position values in [mm] and orientation values in [deg]
- a dict of the form {'x': x, 'y': y, 'z': z, 'roll': roll, 'pitch': pitch, 'yaw': yaw} with position values in [mm] and orientation values in [deg]
- a Coordinates object (copying another Coordinates object)

Constructor:

Create a Coordinates object from coordinate values.

```
coordinates = parse_coordinates({"x": 100, "y": 200, "z": 300,  
                                "roll": 15, "pitch": -15, "yaw": 30})  
coordinates = parse_coordinates([100, 200, 300, 15, -15, 30])
```

```
transform_coordinates(coordinates, relative_tool_frame=None,  
relative_work_frame=None)
```

Transforming the robot coordinates between different frames using their difference in relative tool and work frames. If neither the relative tool frame nor the relative work frame is specified, the coordinates are not being transformed, returning the unchanged coordinates object.

Parameters:

- **coordinates** ([Coordinates](#), list, or dict) –
The TCP coordinates to transform. Specify the value as follows:
 - a Coordinates object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
 - a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}
- **relative_tool_frame** ([Coordinates](#), list, dict, or None) –
The transform to alter the TCP of the robot, given relative to the flange frame. Specify the value as follows:
 - a Coordinates object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
 - a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}

By default, the global tool frame is applied. If the tool frame contains only zero coordinates, it coincides with the flange frame.
- **relative_work_frame** ([Coordinates](#), list, dict, or None) –
The transform to alter the reference of the robot, given relative to the base frame. Specify the value as follows:
 - a Coordinates object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
 - a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}

By default, the global work frame is applied. If the work frame contains only zero coordinates, it coincides with the base frame.

Returns:

The transformed robot coordinates.

Return type:

[Coordinates](#)

Examples:

Transforming a set of coordinates into another work frame. If the origin of the work frame is exactly where the coordinates to transform are, the transformation returns a coordinates object with zeros for all values.

```
coordinates = [150, -250, 600, 15, -60, -15]  
transformed_coordinates = transform_coordinates(coordinates,  
relative_work_frame=coordinates)
```

get_grid_points(*grid_name*)

Reading the positions of all grid points from a specific grid previously saved in the database.

Parameters:

grid_name (*str*) – The name of the grid to extract points from.

Returns:

The list of grid points, in the iteration order defined in the grid properties.

Return type:

list

Examples:

Moving to all points in a predefined grid.

```
# defining the orientation with which to approach the grid points
orientation = [0.0, -85.0, 0.0]
# iterate through all grid points
for position in get_grid_points("sample_tray"):
    # move to each of the grid points
    coordinates = position + orientation
    move_coordinates(coordinates)
```

```
forward_kinematics(joint_angles, tool_offset=None, tool_frame=None,  
work_offset=None, work_frame=None)
```

Forward kinematics is a concept in robot kinematics that computes the pose of the TCP (tool center point) from a given set of joint angles. The reverse of this function is the *inverse kinematics* function, which calculates the joint angles given the TCP pose of the robot. Kinematic calculations are usually done between the built-in base and flange frame of the robots. The user may however alter the reference frames by defining the work and tool frames. By default, the globally defined work and tool frames are used.

Parameters:

- **joint_angles** (*list of float*) – The list of six joint angles in [deg]
- **tool_offset** ([Coordinates](#), *list, dict, or None*) –
The transform to shift the TCP of the robot, given relative to the tool frame.
Specify the value as follows:
 - a [Coordinates](#) object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
 - a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}By default, there is no tool offset, such that the TCP lies at the origin of the tool frame.
- **tool_frame** ([Coordinates](#), *list, dict, or None*) –
The transform to alter the TCP of the robot, given relative to the flange frame.
Specify the value as follows:
 - a [Coordinates](#) object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
 - a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}By default, the global tool frame is applied. If the tool frame contains only zero coordinates, it coincides with the flange frame.
- **work_offset** ([Coordinates](#), *list, dict, or None*) –
The transform to shift the reference of the robot, given relative to the work frame. Specify the value as follows:
 - a [Coordinates](#) object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
 - a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}By default, there is no work offset, such that the reference lies at the origin of the work frame.
- **work_frame** ([Coordinates](#), *list, dict, or None*) –
The transform to alter the reference of the robot, given relative to the base frame. Specify the value as follows:
 - a [Coordinates](#) object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
 - a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}

By default, the global work frame is applied. If the work frame contains only zero coordinates, it coincides with the base frame.

Returns:

The robot pose object.

Return type:

Pose

Examples:

In a singularity, multiple sets of joint angles lead to the same TCP pose. In this example, both joint angles result in the very same pose.

```
pose_1 = forward_kinematics([0, 0, 0, 0, 0, 0])
pose_2 = forward_kinematics([45, 0, 0, 45, 0, -90])
```

Reading the current TCP pose using two different approaches.

```
pose = forward_kinematics(read_actuator_position())
# ... this is similar to ...
pose = get_pose()
```

```
inverse_kinematics(pose, tool_offset=None, tool_frame=None, work_offset=None,
work_frame=None)
```

Inverse kinematics is a concept in robot kinematics that computes a set of joint angles from the pose of the TCP (tool center point). The reverse of this function is the *forward kinematics* function, which calculates the TCP pose from the joint angles of the robot. Note that *inverse kinematics* usually has multiple solutions, but just returns one of these solutions.

Parameters:

- **pose** (*Pose, str, or int*) –
The pose for which to calculate the corresponding joint angles. Specify the value as follows:
 - a Pose object
 - the name of a predefined pose (saved in the database)
 - the ID of a predefined pose (saved in the database)
- **tool_offset** (*Coordinates, list, dict, or None*) –
The transform to shift the TCP of the robot, given relative to the tool frame. Specify the value as follows:
 - a Coordinates object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
 - a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}

By default, there is no tool offset, such that the TCP lies at the origin of the tool frame.
- **tool_frame** (*Coordinates, list, dict, or None*) –
The transform to alter the TCP of the robot, given relative to the flange frame. Specify the value as follows:
 - a Coordinates object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
 - a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}

By default, the global tool frame is applied. If the tool frame contains only zero coordinates, it coincides with the flange frame.
- **work_offset** (*Coordinates, list, dict, or None*) –
The transform to shift the reference of the robot, given relative to the work frame. Specify the value as follows:
 - a Coordinates object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
 - a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}

By default, there is no work offset, such that the reference lies at the origin of the work frame.
- **work_frame** (*Coordinates, list, dict, or None*) –
The transform to alter the reference of the robot, given relative to the base frame. Specify the value as follows:
 - a Coordinates object

- a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
- a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}

By default, the global work frame is applied. If the work frame contains only zero coordinates, it coincides with the base frame.

Returns:

The list of six joint angles in [deg]

Return type:

list of float

Examples:

Calculating the joint angles of a previously saved pose in the database.

```
inverse_kinematics("grasping_pose")
```

Calculate the joint angles that reach a specific set of coordinates in a specific configuration.

```
inverse_kinematics(create_pose([350.0, 0.0, 550.0, 0.0, -30.0, 0.0], "c1"))
```

```
inverse_kinematics_nearest(coordinates, previous_angles=None, tool_offset=None,  
tool_frame=None, work_offset=None, work_frame=None)
```

This function calculates the solution of the *inverse kinematics* (see function `inverse_kinematics` for details) that is nearest to either the current robot pose, or nearest to a specific pose.

Parameters:

- **coordinates** (*Coordinates, list, or dict*) –
The TCP coordinates for which to calculate the corresponding joint angles.
Specify the value as follows:
 - a Coordinates object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
 - a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}
- **previous_angles** (*list or None*) –
The angles to which the nearest solution will be calculated. Specify the value as follows:
 - None, to use the current robot angles
 - a list of joint angles [deg]
- **tool_offset** (*Coordinates, list, dict, or None*) –
The transform to shift the TCP of the robot, given relative to the tool frame.
Specify the value as follows:
 - a Coordinates object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
 - a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}

By default, there is no tool offset, such that the TCP lies at the origin of the tool frame.
- **tool_frame** (*Coordinates, list, dict, or None*) –
The transform to alter the TCP of the robot, given relative to the flange frame.
Specify the value as follows:
 - a Coordinates object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
 - a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}

By default, the global tool frame is applied. If the tool frame contains only zero coordinates, it coincides with the flange frame.
- **work_offset** (*Coordinates, list, dict, or None*) –
The transform to shift the reference of the robot, given relative to the work frame. Specify the value as follows:
 - a Coordinates object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
 - a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}

By default, there is no work offset, such that the reference lies at the origin of the work frame.

- **work_frame** (*Coordinates, list, dict, or None*) –
The transform to alter the reference of the robot, given relative to the base frame. Specify the value as follows:
 - a Coordinates object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
 - a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}

By default, the global work frame is applied. If the work frame contains only zero coordinates, it coincides with the base frame.

Returns:

The list of six joint angles in [deg]

Return type:

list of float

Examples:

The inverse kinematics can be used to simply check if a specific set of coordinates returns a kinematic solution.

```
inverse_kinematics_nearest([600, 0, 300, 180, -90, 0])
```

In order to get multiple solutions for the same pose, the previous pose statement can be changed accordingly. In this example, two solutions for a standing upright robot, slightly shifted to the front and downwards, are calculated.

```
solution_1 = inverse_kinematics_nearest({"x": 485, "y": 188, "z": 1059,  
"roll": 44, "pitch": -45, "yaw": -81},  
                                         previous_angles=[-160, -20, -20, 20,  
-20, -160])  
solution_2 = inverse_kinematics_nearest([600, 0, 300, 180, -90, 0],  
                                         previous_angles=[20, 15, 30, 30, 15,  
10])
```

```
inverse_kinematics_complete(coordinates, tool_offset=None, tool_frame=None,  
work_offset=None, work_frame=None)
```

This function calculates the solution of the *inverse kinematics* (see function `inverse_kinematics` for details) that is nearest to either the current robot pose, or nearest to a specific pose.

Parameters:

- **coordinates** (*Coordinates, list, or dict*) –
The TCP coordinates for which to calculate the corresponding joint angles.
Specify the value as follows:
 - a Coordinates object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
 - a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}
- **tool_offset** (*Coordinates, list, dict, or None*) –
The transform to shift the TCP of the robot, given relative to the tool frame.
Specify the value as follows:
 - a Coordinates object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
 - a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}

By default, there is no tool offset, such that the TCP lies at the origin of the tool frame.
- **tool_frame** (*Coordinates, list, dict, or None*) –
The transform to alter the TCP of the robot, given relative to the flange frame.
Specify the value as follows:
 - a Coordinates object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
 - a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}

By default, the global tool frame is applied. If the tool frame contains only zero coordinates, it coincides with the flange frame.
- **work_offset** (*Coordinates, list, dict, or None*) –
The transform to shift the reference of the robot, given relative to the work frame. Specify the value as follows:
 - a Coordinates object
 - a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
 - a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}

By default, there is no work offset, such that the reference lies at the origin of the work frame.
- **work_frame** (*Coordinates, list, dict, or None*) –
The transform to alter the reference of the robot, given relative to the base frame. Specify the value as follows:
 - a Coordinates object

- a list of Cartesian coordinates in the format [x, y, z, roll, pitch, yaw]
- a dict of Cartesian coordinates in the format {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}

By default, the global work frame is applied. If the work frame contains only zero coordinates, it coincides with the base frame.

Returns:

The lists of six joint angles in [deg] and their configurations in the format {'c1': [...], 'c2': [...], ...}

Return type:

dict

Examples:

In this example, the robot moves to all sets of joint angles that reach the same TCP coordinates.

```
solutions = inverse_kinematics_complete([600, 0, 300, 180, -90, 0])
for joint_angles in solutions.values():
    move_joints([1, 2, 3, 4, 5, 6], joint_angles)
```

If we know one set of joint angles that reaches a desired pose, this way we can calculate all other solutions that reach the same pose but in different joint configurations.

```
pose = forward_kinematics([0, 90, -90, 0, 0, 0])
solutions = inverse_kinematics_complete(pose.coordinates)
```

wait(seconds)

Waiting and therefore actively blocking the further execution of the code for a defined amount of time.

Parameters:

seconds (*float*) – The duration to wait in [s].

Examples:

This example moves the robot to predefined poses and waits for 3 [s] in-between the movements.

```
move_to_pose("pose_1")
wait(3)
move_to_pose("pose_2")
```

This example repeats some code of yours forever. To make sure the execution speed is limited, the code waits for 100 [ms] at the end of each iteration.

```
while True:
    # enter your code here
    wait(0.1)
```

This example is similar to the example above, but instead of waiting for a constant amount of time, the waiting time is chosen dynamically in order to be able to execute your code in a constant interval every second.

```
import time
cycle_time = 1.0
time_start = time.perf_counter()
while True:
    # enter your code here
    time_now = time.perf_counter()
    elapsed_duration = time_now - time_start
    wait(cycle_time - elapsed_duration)
```

run_script(*script_name*, *arguments=None*)

Running a (blocking) application from within the main application.

Parameters:

- **script_name** (*str*) – The name of the application to run. If the application is in a folder, use the full path to the application (e.g. *folder/subfolder/app*)
- **arguments** (*dict*) – The arguments passed down to the application to run. You may read them in the called application using the built-in *script_arguments* variable.

Examples:

Let's assume that we have a separate application that measures the temperature of the environment. This example calls this application repeatedly, once every minute.

```
cycle_time = 60.0
time_measurement = time.perf_counter()
run_script("measurements/measure_temperature")
while True:
    if time.perf_counter() - time_measurement > cycle_time:
        run_script("measurements/measure_temperature")
        time_measurement += cycle_time
    else:
        wait(0.1)
```

start_parallel_script(*script_name*)

Running a (non-blocking) application in parallel to the main application. Note that the main application continues immediately, and that all parallel applications abort once the main application terminates.

Parameters:

script_name (*str*) – The name of the parallel application to start. If the application is in a folder, use the full path to the application (e.g. *folder/subfolder/app*)

Examples:

This example triggers a parallel application with the purpose of monitoring some temperature sensor to make sure the robot only operates in a specific temperature range. The main application runs continuously, and aborts if the operational temperature criterion is no longer fulfilled.

```
start_parallel_script("measurements/measure_temperature_continuously")
set_global_variable("temperature_value") = 20
while -10 < get_global_variable("temperature_value") < 40:
    move_to_pose("pose_1")
    move_to_pose("pose_2")
    move_to_pose("pose_3")
    move_to_pose("pose_4")
print("Aborting the program, because the measured temperature is outside of
the operational temperature range.")
```

stop_application()

Stopping the currently running applications. This includes the main application, and any applications that have been triggered by the main application. Note that a running *background* application keeps running.

Raises:

ScriptInterrupt – Raised into the main application if running this function from an application that was triggered using the *run_script* function.

Examples:

This code sample can be used in a *background* application to temporarily pause and manually resume the running main applications.

```
pause_application()
answer = dialog_choice("Do you want to continue?", ["Continue", "Abort"],
title="Application Interrupt", dialog_value="Abort")
if answer == "Continue":
    resume_application()
else:
    stop_application()
```

pause_application()

Pausing the currently running applications. This includes the main application, and any applications that have been triggered by the main application. Note that a running *background* application keeps running.

Examples:

Have a look at the combined example of *stop_application*, which uses this function.

resume_application()

Resuming the currently pausing applications. Since this function can only be run in *paused* state, it can only be used from within a *background* application.

Examples:

Have a look at the combined example of *stop_application*, which uses this function.

stop_and_release()

Stopping the currently running applications, and releasing all joints (enabling the gravity compensation mode). Note that the user needs to manually hold all joints (disabling the gravity compensation mode) to continue running further applications afterward.

is_status(*status*)

Checking whether the program status corresponds to the given status.

Parameters:

status (*str or list of str*) –

The program status to check for, can be one or multiple of the following:

- 'ready' (the robot is idle and waiting to receive commands)
- 'processing' (the robot is processing data which may take a while)
- 'hand_guidance_control' (the robot is in hand-guidance control mode)
- 'running' (the robot is running an application or motion)
- 'paused' (the robot is pausing an application or motion)
- 'emergency_stop' (emergency stop is triggered)
- 'normal_stop' (normal stop is triggered)
- 'protective_stop' (protective stop is triggered)
- 'error' (an error or safety violation occurred)
- 'position_limit' (a safe limited position violation occurred)
- 'safeguard' (a safeguard interrupt is triggered)
- 'collision' (a collision interrupt is triggered)
- 'proceed' (the robot is proceeding after an interrupt)
- 'stopping' (the robot is stopping its program)

Returns:

Whether the program status matches the given status (or is one of the given status)

Return type:

bool

```
dialog_choice(text, options, title="", dialog_type=0, dialog_value=None,  
mandatory=False)
```

Creating a dialog with buttons. This dialog blocks the program flow until the user answers the dialog with one of the presented options.

Parameters:

- **text** (*str*) – The message (or question) presented to the user.
- **options** (*list of str*) – The options that can be chosen, which are the labels on each button.
- **title** (*str*) – The title of the dialog pop-up.
- **dialog_type** (*int*) –
The type of dialog to present to the user. This can be one of the following:
 - 0: information / blue color
 - 1: success / green color
 - 2: warning / orange color
 - 3: error / red color
- **dialog_value** (*str*) – The default value to return if the dialog is not answered.
- **mandatory** (*bool*) – Whether the dialog must be answered for the program flow to continue. If set to *True*, the dialog cannot be ignored by e.g. closing the pop-up window, and it blocks until an answer is received.

Returns:

The selected option, which is the label on the button pressed by the user.

Return type:

str

Examples:

This example shows how to create a simple informative pop-up window containing a "Continue" button. Upon pressing the button, the program will continue executing its code.

```
dialog_choice("This is just an informative message.", ["Continue"],  
title="Information", dialog_type=0)
```

```
dialog_dropdown(text, options, title='', dialog_type=0, dialog_value=None,  
mandatory=False)
```

Creating a dialog with options presented in a drop-down list. The dialog blocks the program flow until the user answers the dialog by selecting and confirming one of the options.

Parameters:

- **text** (*str*) – The message (or question) presented to the user.
- **options** (*list of str*) – The options that can be chosen, which are the labels of the drop-down options.
- **title** (*str*) – The title of the dialog pop-up.
- **dialog_type** (*int*) –
The type of dialog to present to the user. This can be one of the following:
 - 0: information / blue color
 - 1: success / green color
 - 2: warning / orange color
 - 3: error / red color
- **dialog_value** (*str*) – The default value to return if the dialog is not answered.
- **mandatory** (*bool*) – Whether the dialog must be answered for the program flow to continue. If set to *True*, the dialog cannot be ignored by e.g. closing the pop-up window, and it blocks until an answer is received.

Returns:

The selected option, which is the label of the selected drop-down option.

Return type:

str

Examples:

This example shows how to create a warning pop-up window and let's the user choose between multiple options to influence the program flow.

```
answer = dialog_dropdown("High temperature detected. You may want to reduce  
the robot's speed.",  
                        options=["Reduce by 20%", "Reduce by 10%", "Do not  
reduce"],  
                        title="High Temperature Warning", dialog_type=2,  
dialog_value="Do not reduce")  
if answer == "Reduce by 20%":  
    velocity *= 0.8  
elif answer == "Reduce by 10%":  
    velocity *= 0.9
```

dialog_text(text, title= '', dialog_type=0, dialog_value=None, mandatory=False)

Creating a dialog with a text field. The dialog blocks the program flow until the user enters a text and confirms the dialog.

Parameters:

- **text** (*str*) – The message (or question) presented to the user.
- **title** (*str*) – The title of the dialog pop-up.
- **dialog_type** (*int*) –
The type of dialog to present to the user. This can be one of the following:
 - 0: information / blue color
 - 1: success / green color
 - 2: warning / orange color
 - 3: error / red color
- **dialog_value** (*str*) – The default value to return if the dialog is not answered.
- **mandatory** (*bool*) – Whether the dialog must be answered for the program flow to continue. If set to *True*, the dialog cannot be ignored by e.g. closing the pop-up window, and it blocks until an answer is received.

Returns:

The text entered into the text field.

Return type:

str

Examples:

This example shows how to create a simple pop-up window with a text field for entering your name. After the user enters their name, it can be further used in the program, e.g. for logging.

```
name = dialog_text("Please enter your name:", title="Welcome!")
log_message("Registered user: {name}")
```

dialog_yes_no(text, title='', dialog_type=0, dialog_value=None, mandatory=False)

Creating a dialog with a Yes and No button. The dialog blocks the program flow until the user presses one of these buttons.

Parameters:

- **text** (*str*) – The message (or question) presented to the user.
- **title** (*str*) – The title of the dialog pop-up.
- **dialog_type** (*int*) –
The type of dialog to present to the user. This can be one of the following:
 - 0: information / blue color
 - 1: success / green color
 - 2: warning / orange color
 - 3: error / red color
- **dialog_value** (*str*) – The default value to return if the dialog is not answered.
- **mandatory** (*bool*) – Whether the dialog must be answered for the program flow to continue. If set to *True*, the dialog cannot be ignored by e.g. closing the pop-up window, and it blocks until an answer is received.

Returns:

True if the user selects "yes", False if the user selects "no".

Return type:

bool

Examples:

This example uses a simple confirmation dialog to let the user choose whether to continue or abort the application.

```
if not dialog_yes_no("An error occurred. Do you want to continue anyway?",
title="Error", dialog_type=3):
    stop_application()
```

print(*args)

Printing a message into the application output. Note that this function is similar in usage to the built-in *print* function of Python. It supports multiple arguments and various argument types. Arguments are automatically converted to strings and concatenated with a white-space separator.

Parameters:

args (*tuple of various*) – The arguments to be printed.

Examples:

Printing the value of a variable in a nicely formatted style. Note that the first call uses automatic conversion and concatenation, while the second call uses f-strings with the same result.

```
print("Variable value:", var)
print(f"Variable value: {var}")
```

`show_notification(message_text, message_title= 'Notification')`

Showing a custom notification message in the interface. This has no effect on the program flow. Whether notifications are fading or sticking in the interface, the user may specify in the settings menu.

Parameters:

- **message_text** (*str*) – The notification text to show in the interface.
- **message_title** (*str*) – The title of the notification pop-up.

raise_error(message)

Raising a custom error message into the running application. The error is of type `ApplicationError` and can be caught in the application. If not caught, the running application aborts. The error is also logged in the `error.log` file, which is part of the logging system. Logs can be downloaded via the interface by pressing `CTRL + SHIFT + L`.

Parameters:

message (*str*) – The error message to raise into the application.

Raises:

ApplicationError – Containing the custom message.

Examples:

A background application monitors digital inputs and checks if a specific error input is being triggered. If an error input is triggered, this error is being raised into the application.

```
# keeping track of the previous error flag value
error_status = False
# do forever
while True:
    # read the value of the error flag (True or False)
    error_flag = read_digital_main_input("error_flag")
    # if the error flag switches from False to True, raise an application
    # error
    if error_flag:
        if not error_status:
            raise_error("The error input was raised!")
            error_status = True
        else:
            error_status = False
    wait(0.1)
```

This application error can then be caught and handled by the main application controlling the robot.

```
try:
    my_main_program()
except ApplicationError as error:
    ...
```

raise_warning(*message*)

Raising a custom warning message box into the interface. This has no effect on the program flow. Whether warnings are fading or sticking in the interface, the user may specify in the settings menu.

Parameters:

message (*str*) – The warning message to raise into the interface.

Examples:

Let's assume we have a temperature sensor attached to an analog input. This example raises a warning into the interface if the value increases above some threshold, and resets if the value decreases again below some threshold.

```
# flag to store whether a warning was raised
high_temperature = False
# do forever
while True:
    # check temperature value
    temperature_value = read_analog_main_input("temperature_sensor")
    # if value is too high, raise a warning
    if temperature_value > 5.2:
        if not high_temperature:
            raise_warning("High temperature detected! Slow down the robot to
prevent overheating.")
            high_temperature = True
        # if value recovers, reset the flag, such that a warning can be raised
again later on
        elif temperature_value < 5.0:
            high_temperature = False
    wait(5)
```

log_message(message, logger='custom', include_timestamp=True)

Logging a message to a custom log file.

Parameters:

- **message** (*str*) – The custom message to log.
- **logger** (*str*) – The name of the log file to log into. The log file can be found at `custom/<logger>.log`, which in the log folder. The default file name is `custom/custom.log`. Logs can be downloaded via the interface by pressing `CTRL + SHIFT + L`.
- **include_timestamp** (*bool*) – Whether a timestamp is added to the logged message.

Examples:

Logging the value of a sensor measurements into the `custom/measurements.log` file.

```
value = read_analog_main_input("my_sensor")
log_message(f"Sensor value: {value}", logger="measurements")
```

clear_log()

Clearing and removing all custom log files.

set_shared_variable(name, value)

Setting the value of a shared variable.

Note that *shared variables* only exist as long as the main application runs. After termination of the main application, all *shared variables* are automatically removed. If you require variables to keep their values across application runtime or even across restarting the robot, use *global variables* instead.

Parameters:

- **name** (*str*) – The name of the variable to set.
- **value** (*various*) – The new value of this variable.

Examples:

This example shows how a parallel application can be terminated by the main application using shared variables. The parallel application simply listens to a termination flag, and aborts as soon as the flag is set to *True*. In addition, the shared variable is removed once it no longer has any effect. Since shared variables are wiped once the main application terminates, this is not necessary, but good practice.

```
while not get_shared_variable("terminate_parallel_application"):
    ...
    wait(0.1)
remove_shared_variable("terminate_parallel_application")
```

The main application makes sure that the termination flag exists, runs the parallel application, and finally sets the termination flag to *True* to finalize the parallel application.

```
set_shared_variable("terminate_parallel_application", False)
start_parallel_script("my_parallel_application")
...
set_shared_variable("terminate_parallel_application", True)
...
```

get_shared_variable(name=None)

Reading the value of a specific shared variable. By default, this function reads the values of all shared variables.

Note that *shared variables* only exist as long as the main application runs. After termination of the main application, all *shared variables* are automatically removed. If you require variables to keep their values across application runtime or even across restarting the robot, use *global variables* instead.

Parameters:

name (*str* (or *None*)) – name of the variable to return (or *None* if all variables are to be returned)

Returns:

value of the variable to return (or dictionary containing all variables)

Return type:

various (or dict)

Examples:

Have a look at the examples of *set_shared_variable*, which combines the usage of all *shared variable* functions.

remove_shared_variable(*name*)

Removing a shared variable. After removing it, its value can no longer be accessed.

Note that *shared variables* only exist as long as the main application runs. After termination of the main application, all *shared variables* are automatically removed. If you require variables to keep their values across application runtime or even across restarting the robot, use *global variables* instead.

Parameters:

name (*str*) – name of the variable to remove

Examples:

Have a look at the examples of *set_shared_variable*, which combines the usage of all *shared variable* functions.

set_global_variable(*name*, *value*)

Setting the value of a global variable.

Note that changes to *global variables* are permanent and survive even shutting down and restarting the robot. Therefore, it is highly recommended to remove *global variables* once they no longer serve any purpose. If you need variables only during runtime of a single application, use *shared variables* instead.

Parameters:

- **name** (*str*) – The name of the variable to set.
- **value** (*various*) – The new value of this variable.

Examples:

This example shows how a repetitive application counts the number of its successful iterations. This is done by initializing a global counter at first runtime, and increasing this counter at the end of each loop iteration.

```
if "counter" not in get_global_variable():
    set_global_variable("counter", 0)
while True:
    ...
    counter = get_global_variable("counter")
    set_global_variable("counter", counter + 1)
```

get_global_variable(name=None)

Reading the value of a specific global variable. By default, this function reads the values of all global variables.

Note that changes to *global variables* are permanent and survive even shutting down and restarting the robot. Therefore, it is highly recommended to remove *global variables* once they no longer serve any purpose. If you need variables only during runtime of a single application, use *shared variables* instead.

Parameters:

name (*str* (or *None*)) – name of the variable to return (or *None* if all variables are to be returned)

Returns:

value of the variable to return (or dictionary containing all variables)

Return type:

various (or dict)

Examples:

Have a look at the examples of *set_global_variable*, which combines the usage of *global variable* functions. functions.

remove_global_variable(*name*)

Removing a global variable. After removing it, its value can no longer be accessed.

Note that changes to *global variables* are permanent and survive even shutting down and restarting the robot. Therefore, it is highly recommended to remove *global variables* once they no longer serve any purpose. If you need variables only during runtime of a single application, use *shared variables* instead.

Parameters:

name (*str*) – name of the variable to remove

Examples:

This example removes all global variables.

```
for name in get_global_variable().keys():  
    remove_global_variable(name)
```

set_state_variable(name, value)

Setting the value of a state variable.

Note that *state variables* are defined for each application individually, and they are only used in main applications. They can be set up and customized in the *variables* tab of the application editor. Each *state variable* has a specific type, and will only accept values of this particular type. The main purpose of *state variables* is that their value is shown live in the *Panel Interface* for observing and controlling the running application.

Parameters:

- **name** (*str*) – The name of the variable to set.
- **value** (*various*) – The new value of this variable.

Examples:

Let's assume that for this application a read-only state variable of type *percent* and name *progress* and a dynamically editable state variable of type *boolean* and name *logging* was defined. The purpose of the *progress* variable is to store the current progress of the program in percent. The purpose of the *logging* flag is whether the progress update is to be logged. Note that the user can anytime flip the value of *logging*, which dynamically adapts the program.

```
number_of_iterations = 128
for i in range(number_of_iterations):
    progress = i / number_of_iterations * 100
    set_state_variable("progress", progress)
    if get_state_variable("logging"):
        log_message(f"The current progress is {progress}%.")
    ...
set_state_variable("progress", 100)
if get_state_variable("logging"):
    log_message(f"The current progress is 100%. The program successfully
completed.")
```

get_state_variable(name=None)

Reading the value of a specific state variable. By default, this function reads the values of all state variables.

Note that *state variables* are defined for each application individually, and they are only used in main applications. They can be set up and customized in the *variables* tab of the application editor. Each *state variable* has a specific type, and will only accept values of this particular type. The main purpose of *state variables* is that their value is shown live in the *Panel Interface* for observing and controlling the running application.

Parameters:

name (*str (or None)*) – name of the variable to return (or None if all variables are to be returned)

Returns:

value of the variable to return (or dictionary containing all variables)

Return type:

various (or dict)

Examples:

Have a look at the examples of *set_state_variable*, which combines the usage of *state variable* functions.

read_actuator_position(*actuator_ids=None*)

Reading the angular position of one, many, or all actuators.

Parameters:

actuator_ids (*None, str, int, list, set*) –

The IDs of the actuators to read from. Specify the value as follows:

- the ID of a single joint (e.g. 6)
- a list of joint IDs for multiple joints (e.g. [1, 4, 6])
- a set of joint names for multiple joints (e.g. {1, 4, 6})
- the constant *ALL_KIN_JOINTS* or *None* to read from all joints

Returns:

The angular position of the requested actuators (in [deg]), which can be:

- a single joint position value -> float
- a list of joint position values -> list of float
- a mapping between joint IDs and their position values -> dict of float

Return type:

float, or list of float, or dict of float

Examples:

Printing the current positions of the wrist joint actuators.

```
print(read_actuator_position([1, 4, 6]))
```

read_actuator_velocity(*actuator_ids=None*)

Reading the angular velocity of one, many, or all actuators.

Parameters:

actuator_ids (*None, str, int, list, set*) –

The IDs of the actuators to read from. Specify the value as follows:

- the ID of a single joint (e.g. 6)
- a list of joint IDs for multiple joints (e.g. [1, 4, 6])
- a set of joint names for multiple joints (e.g. {1, 4, 6})
- the constant *ALL_KIN_JOINTS* or *None* to read from all joints

Returns:

The angular velocity of the requested actuators (in °/s), which can be:

- a single joint velocity value -> float
- a list of joint velocity values -> list of float
- a mapping between joint IDs and their velocity values -> dict of float

Return type:

float, or list of float, or dict of float

Examples:

Monitoring the velocity during a movement to determine the fastest moving actuator.

```
move_to_pose("target", block=False)
max_velocities = {1: 0, 2: 0, 3: 0, 4: 0, 5: 0, 6: 0}
while is_motor_moving():
    velocities = read_actuator_velocities({1, 2, 3, 4, 5, 6})
    for joint in range(1, 7):
        max_velocities[joint] = max([max_velocities[joint],
abs(velocities[joint])])
fastest_joint = max(max_velocities, key=max_velocities.get)
print(f"The fastest joint is {fastest_joint} with a maximum velocity of
{max_velocities[fastest_joint]}°/s")
```

get_linear_velocity()

Getting the linear velocity of the TCP (tool center point) of the robot.

Returns:

The linear tool velocity in [mm/s].

Return type:

list

Examples:

Get and print the current linear velocity.

```
print(get_linear_velocity())
```

get_angular_velocity()

Getting the angular velocity of the TCP (tool center point) of the robot.

Returns:

The angular tool velocity in [deg/s].

Return type:

list

Examples:

Get and print the current angular velocity.

```
print(get_angular_velocity())
```

read_actuator_current(*actuator_ids=None*)

Reading the applied current in one, many, or all actuators.

Parameters:

actuator_ids (*None, str, int, list, set*) –

The IDs of the actuators to read from. Specify the value as follows:

- the ID of a single joint (e.g. 6)
- a list of joint IDs for multiple joints (e.g. [1, 4, 6])
- a set of joint names for multiple joints (e.g. {1, 4, 6})
- the constant *ALL_KIN_JOINTS* or *None* to read from all joints

Returns:

The current values of the requested actuators (in A), which can be:

- a single joint current value -> float
- a list of joint current values -> list of float
- a mapping between joint IDs and their current values -> dict of float

Return type:

float, or list of float, or dict of float

Examples:

Printing the currents of the robot actuators at a specific pose twice, once before and once after picking an object.

```
move_to_pose("measure_pose")
print(f"Currents without payload: {read_actuator_current()}")
move_to_pose("pick_pose")
tool_pick()
move_to_pose("measure_pose")
print(f"Currents with payload: {read_actuator_current()}")
```

tool_pick()

Activating the tool's pick function to grasp an object. This may result in clamping for mechanical tools or applying suction for pneumatic tools.

Examples:

Activate the tool pick function.

```
tool_pick()
```

tool_place()

Activating the tool's place function to release an object. This may result in opening a clamp for mechanical tools or stopping suction for pneumatic tools.

Examples:

Activate the tool place function.

```
tool_place()
```

set_tool_payload(payload, center_of_mass=None, inertia=None)

Set the tool payload dynamically during robot operation.

Parameters:

- **payload** (*float*) – the tool payload (weight + grabbed items) in [kg]
- **center_of_mass** (*list of float*) – center of mass [x, y, z] relative to the flange frame in [m]
- **inertia** (*list of float*) – inertia matrix sliced into list with length 9 [kg m²]

Examples:

Changing the payload while picking and placing objects from one pose to another.

```
tool_mass = 1.3
object_mass = 0.7

while True:
    move_pose("pick_pose")
    tool_pick()
    set_tool_payload(tool_mass + object_mass)

    move_pose("place_pose")
    tool_place()
    set_tool_payload(tool_mass)
```

read_digital_main_input(*identifiers*)

Reading the value of one, many, or all I/Os of the 'Digital Main Inputs' category.

The two states of digital I/Os are:

- *HIGH*, which is represented by the boolean value *True*
- *LOW*, which is represented by the boolean value *False*

Parameters:

identifiers (*int, str, list (of int or str), set (of int or str)*) –

Specifies which I/Os to read. Identifiers are either the I/O numbers, or the unique custom names that can be assigned to these I/Os. This can be one of the following:

- a single number (int) or custom name (str) -> returns a bool
- a list of numbers (int) or custom names (str), or mixed -> returns a list of bool
- a set of numbers (int) or custom names (str), or mixed -> returns a dict with bool values and identifiers as keys

Returns:

The boolean values of the requested digital I/Os (HIGH = True, LOW = False)

Return type:

bool, list, dict

Examples:

Let's assume that the name 'trigger' was assigned to one of the digital inputs. This code block waits until this trigger switches to HIGH before continuing its execution.

```
while not read_digital_main_input("trigger"):
    wait(0.1)
```

Reading the first 8 digital I/Os and converting the binary list of bools to an integer number between 0 and 255.

```
bool_list = read_digital_main_input(list(range(1, 9)))
int_representation = int(''.join([str(int(bit)) for bit in bool_list]), 2)
```

read_digital_tool_input(*identifiers*)

Reading the value of one, many, or all I/Os of the 'Digital Tool Inputs' category.

For details, refer to 'read_digital_main_input', which is similar in usage.

read_analog_main_input(*identifiers*)

Reading the value of one, many, or all I/Os of the 'Analog Main Inputs' category.

Analog I/Os can either be in voltage or current mode. Values are applied and read in steps of 0.1, meaning that the I/O value 55 represents the real voltage value 5.5 V or current value 5.5 mA. The value range of analog I/Os is:

- Between 0.0 and 10.0 V (in voltage mode)
- Between 4.0 and 20.0 mA (in current mode)

Parameters:

identifiers (*int, str, list (of int or str), set (of int or str)*) –

Specifies which I/Os to read. Identifiers are either the I/O numbers, or the unique custom names that can be assigned to these I/Os. This can be one of the following:

- a single number (int) or custom name (str) -> returns a bool
- a list of numbers (int) or custom names (str), or mixed -> returns a list of bool
- a set of numbers (int) or custom names (str), or mixed -> returns a dict with bool values and identifiers as keys

Returns:

The numeric values of the requested analog I/Os

Return type:

float, list, dict

Examples:

Let's assume that the name 'temperature' was assigned to one of the analog inputs in current mode. A temperature sensor was attached to this input, and it was determined that the return value of 10.6 mA represents a critical temperature threshold that should not be exceeded. Then run the following program in parallel to your main application, which aborts the program if the critical threshold was reached.

```
threshold = 10.6
while True:
    value = read_analog_main_input("temperature")
    if value >= threshold:
        raise_error("High temperature detected -> aborting the program")
    wait(0.1)
```

read_analog_tool_io(*identifiers*)

Reading the value of one, many, or all I/Os of the 'Analog Tool I/O' category.

For details, refer to 'read_analog_main_input', which is similar in usage.

write_digital_main_output(*identifiers, values*)

Writing the value of one, many, or all I/Os of the 'Digital Main Output' category.

The two states of digital I/Os are:

- *HIGH*, which is represented by the boolean value *True*
- *LOW*, which is represented by the boolean value *False*

Parameters:

- **identifiers** (*int, str, list (of int or str), set (of int or str)*) –
Specifies which I/Os to write. Identifiers are either the I/O numbers, or the unique custom names that can be assigned to these I/Os. This can be one of the following:
 - a single number (int) or custom name (str) -> *values* is a bool
 - a list of numbers (int) or custom names (str), or mixed -> *values* is a list of bool
 - a set of numbers (int) or custom names (str), or mixed -> *values* is a dict with bool values and identifiers as keys
- **values** (*bool, list (of bool), dict (of bool)*) –
The boolean values of the digital I/Os to write (HIGH = True, LOW = False). Depending on the *identifiers*, this can be one of the following:
 - a single bool -> if *identifiers* is a single I/O
 - a list of bool -> if *identifiers* is a list of I/Os
 - a dict with bool values and identifies as keys -> if *identifiers* is a set of I/Os

Examples:

Let's assume that the name 'trigger' was assigned to one of the digital output. This code sends an impulse to this digital output by setting it to HIGH for 100 ms, and then back to LOW.

```
write_digital_main_output("trigger", True)
wait(0.1)
write_digital_main_output("trigger", False)
```

Writing an 8-bit integer number (between 0 and 255) to the first 8 digital I/Os, by converting the number to a bool list first, and then writing all 8 values simultaneously.

```
number = 123
bool_list = [bool(int(bit)) for bit in f"{number:08b}"]
write_digital_main_output(list(range(1, 9)), bool_list)
```

`write_digital_tool_output(identifiers, values)`

Writing the value of one, many, or all I/Os of the 'Digital Tool Output' category.

For details, refer to 'write_digital_main_output', which is similar in usage.

write_analog_main_output(*identifiers, values*)

Writing the value of one, many, or all I/Os of the 'Analog Main Outputs' category.

Analog I/Os can either be in voltage or current mode. Values are applied and read in steps of 0.1, meaning that the I/O value 55 represents the real voltage value 5.5 V or current value 5.5 mA. The value range of analog I/Os is:

- Between 0.0 and 10.0 V (in voltage mode)
- Between 4.0 and 20.0 mA (in current mode)

Parameters:

- **identifiers** (*int, str, list (of int or str), set (of int or str)*) –
Specifies which I/Os to write. Identifiers are either the I/O numbers, or the unique custom names that can be assigned to these I/Os. This can be one of the following:
 - a single number (int) or custom name (str) -> *values* is a float
 - a list of numbers (int) or custom names (str), or mixed -> *values* is a list of float
 - a set of numbers (int) or custom names (str), or mixed -> *values* is a dict with float values and identifiers as keys
- **values** (*float, list (of float), dict (of float)*) –
The numeric values of the analog I/Os to write. Depending on the *identifiers*, this can be one of the following:
 - a single float -> if *identifiers* is a single I/O
 - a list of float -> if *identifiers* is a list of I/Os
 - a dict with float values and identifies as keys -> if *identifiers* is a set of I/Os

Examples:

Let's assume that the R (red), G (green), and B (blue) values of a controllable LED are directly connected to three analog I/Os with custom names "red", "green" and "blue", such that the maximum value range [0, 255] directly maps to the maximum voltage range [0.0 V, 10.0 V]. This example writes these values simultaneously.

```
if color == "orange":
    color_code = [255, 188, 32]
elif color == "green":
    color_code = [31, 226, 0]
elif color == "purple":
    color_code = [170, 22, 218]
else:
    raise_error("Unknown color")
write_analog_main_output(["red", "green", "blue"], [c/2.55 for c in
color_code])
```

`write_analog_tool_io(identifiers, values)`

Writing the value of one, many, or all I/Os of the 'Analog Tool I/O' category.

For details, refer to 'write_analog_main_output', which is similar in usage.

wait_for_digital_main_input(*identifier*, *value*)

Waiting until an I/O of the 'Digital Main Inputs' category reads a specific value.

The two states of digital I/Os are:

- *HIGH*, which is represented by the boolean value *True*
- *LOW*, which is represented by the boolean value *False*

Parameters:

- **identifier** (*int*, *str*) – Specifies which I/O to wait for. The identifier is either the I/O number (*int*), or the unique custom name (*str*) that can be assigned to this I/O.
- **value** (*bool*) – The value to wait for.

Examples:

Let's assume that the name 'trigger' was assigned to one of the digital inputs. This code block waits until this trigger switches to HIGH before continuing its execution.

```
wait_for_digital_main_input("trigger", True)
```

In rare cases you may want to be able to pause a program and you still want the waiting function to terminate if the I/O is triggered. This waiting function cannot be used for this approach, but the following example does exactly that:

Add this code into a background application. It monitors the value of a specific I/O and sets a flag in case the specified value was reached while monitoring.

```
# writing the 'trigger' value into a global variable for further processing
set_global_variable("triggered_flag", False)
while True:
    if not get_global_variable("triggered_flag"):
        set_global_variable("triggered_flag",
read_digital_main_input("trigger"))
        wait(0.01)
```

And add this code to the main application counterpart that triggers the background monitor and waits until the flag is set by the monitor.

```
# waiting for the 'trigger' value to be written into a global variable
set_global_variable ("triggered_flag", False)
while not get_global_variable("triggered_flag"):
    wait(0.01)
```

wait_for_digital_tool_input(*identifier, value*)

Waiting until an I/O of the 'Digital Tool Inputs' category reads a specific value.

For details, refer to 'wait_for_digital_main_input', which is similar in usage.

wait_for_analog_main_input(*identifier*, *value_range*)

Waiting until an I/O of the 'Analog Main Inputs' category exceeds a threshold.

Analog I/Os can either be in voltage or current mode. Values are applied and read in steps of 0.1, meaning that the I/O value 55 represents the real voltage value 5.5 V or current value 5.5 mA. The value range of analog I/Os is:

- Between 0.0 and 10.0 V (in voltage mode)
- Between 4.0 and 20.0 mA (in current mode)

Parameters:

- **identifier** (*int*, *str*) – Specifies which I/O to wait for. The identifier is either the I/O number (*int*), or the unique custom name (*str*) that can be assigned to this I/O.
- **value_range** (*list of float*) –
Waiting for this I/O to reach a value within this range. This is one of the following:
 - both lower and upper limit (e.g. [5.5, 7.0])
 - only a lower limit (e.g. [5.5, None])
 - only an upper limit (e.g. [None, 7.0])

Examples:

Let's consider an analog I/O in voltage mode. This example waits for its voltage value to grow beyond 8.0 V, which is represented by the value 80 (in steps of 0.1 V).

```
wait_for_analog_main_input(6, [80, None])
```

`wait_for_analog_tool_io(identifier, value_range)`

Waiting until an I/O of the 'Analog Tool I/O' category exceeds a threshold.

For details, refer to 'wait_for_analog_main_input', which is similar in usage.

get_runtime_position_offsets()

This function returns the position offsets of the robot.

Returns:

position offsets of the robot

Return type:

list of float

read_button_state(channel=None)

Reading the current state (pressed or not) of one or all customizable buttons of the robot's interface. Note that the buttons with specific assigned functionality cannot be read on run-time, as they are used otherwise.

These buttons are located at the top most joint of the robot arm.

Parameters:

channel (*int or None*) – The button identifier, which is a number between 1 and the number of buttons. By default, the states of all customizable buttons are read.

Returns:

The current state of the requested button as a bool, or the current state of each button, packed into a dictionary of the form {1: bool, 2: bool, ...}.

Return type:

bool or dict of bool

Examples:

This example prints whenever the state of a customizable button changes.

```
button_states = read_button_state()
while True:
    for channel, value in read_button_state().items():
        if not button_states[channel] and value:
            print(f"Button {channel} was pressed.")
        elif button_states[channel] and not value:
            print(f"Button {channel} was released.")
        button_states[channel] = value
    wait(0.1)
```

Run this example as a background application. It allows you to pause and resume the running main application using the buttons 1 and 2.

```
pausing = False
while True:
    if read_button_state(1) and not pausing:
        pause_application()
        pausing = True
    elif read_button_state(2) and pausing:
        resume_application()
        pausing = False
    wait(0.1)
```

create_tcp_client(*ip*, *port*)

Creating a TCP socket client for transmitting messages. The client continuously tries to connect to the server at the specified address. Messages are encoded with the *UTF-8* encoding standard. Furthermore, messages are separated by the end-of-text character *0x03* (`\x03` in Python).

Parameters:

- **ip** (*str*) – The IP address of the server (e.g. `"192.168.80.2"`)
- **port** (*int*) – The port number through which the server is accessible (e.g. `60001`). Ports are limited to the range between 60000 and 61000.

Returns:

The TCP socket client object, which is used to send and receive messages.

Return type:

Socket

Examples:

This example performs a handshake between a TCP socket client in the application and a TCP socket server in the same network. First, the connection is established. Then the client waits for the server to send an initial message. Upon receiving this one, the client responds with its own message, and finally closes the connection.

```
client = create_tcp_client("192.168.80.2", 60001)
print(client.receive())
client.send("I am your client")
client.close()
```

create_tcp_server(*port*)

Creating a TCP socket server that can handle up to 20 clients. Messages are encoded with the *UTF-8* encoding standard. Furthermore, messages are separated by the end-of-text character *0x03* (`\x03` in Python).

Parameters:

port (*int*) – The port number through which this server can be accessed by clients (e.g. `60002`). Ports are limited to the range between 60000 and 61000.

Returns:

The TCP socket server object, which is used to send and receive messages.

Return type:

Socket

Examples:

This example performs a handshake between a TCP socket server in the application and the first client that connects to it. First, the server opens the connection on a specific port and waits for a client to connect. Then the server waits for the client to send an initial message. Upon receiving this one, the server responds with its own message, and finally closes the connection to all connected clients.

```
server = create_tcp_server(60002)
print(server.receive())
server.send("I am your server")
server.close()
```

create_udp_socket(port)

Creating a UDP socket for transmitting messages. Messages are encoded with the *UTF-8* encoding standard. Furthermore, messages are separated by the end-of-text character *0x03* (`\x03` in Python).

Parameters:

port (*int*) – The port number through which this socket can be accessed (e.g. `60003`). Ports are limited to the range between 60000 and 61000.

Returns:

The UDP socket object, which is used to send and receive messages.

Return type:

[Socket](#)

Examples:

This example performs a handshake between a UDP socket in the application and the first connection that opens to it. First, the socket opens the connection on a specific port and waits for some counterpart to connect. Then the socket waits for the counterpart to send an initial message. Upon receiving this one, the socket responds with its own message, and finally closes the connection.

```
udp_socket = create_udp_socket(60003)
print(udp_socket.receive())
udp_socket.send("I am your socket")
udp_socket.close()
```

`send_message(message)`

Sending a message with key 'script_send' to an open TCP connection.

Parameters:

message (*str*) – The message to send.

Examples:

Sending a message via TCP to any device that may listen to it:

```
send_message("Hello World!")
```

On the connected device side, this data is received in the following format:

```
{"script_send": "Hello World!"}
```

receive_message(*timeout=None*)

Receiving a message from an open TCP connection calling the action 'script_receive'.

Parameters:

timeout (*None or float*) – The maximum time to wait for a message. If not specified, wait indefinitely.

Examples:

An external device connects to the robot via TCP and sends the following data:

```
{"bridge": "core", "action": "script_receive", "message": "Hello World!"}
```

On robot side, this function can then be used to receive and print the message:

```
message = receive_message(timeout=3.0)
print(message)
```

clear_messages()

Clearing the TCP message queue from previously incoming messages that might not have been read.

Examples:

Before waiting for a new message, sometimes it makes sense to make sure that old messages are rejected.

```
clear_message()  
message = receive_message()
```

***class* Coordinates**

The Coordinates object defines a mathematical transform that can be used e.g. as target coordinates in movement functions, or to define coordinate frames.

`__init__(coordinates)`

Create a Coordinates object, either with specific coordinate values, or containing the current robot coordinates.

Parameters:

coordinates (*list, dict, or [Coordinates](#)*) –

A specific set of coordinates, which can be one of the following:

- a list of the form [x, y, z, roll, pitch, yaw] with position values in [mm] and orientation values in [deg]
- a dict of the form {'x': x, 'y': y, 'z': z, 'roll': roll, 'pitch': pitch, 'yaw': yaw} with position values in [mm] and orientation values in [deg]
- a Coordinates object (copying another Coordinates object)

Constructor:

Create a Coordinates object from coordinate values.

```
coordinates = Coordinates({"x": 100, "y": 200, "z": 300,  
                           "roll": 15, "pitch": -15, "yaw": 30})  
coordinates = Coordinates([100, 200, 300, 15, -15, 30])
```

Create a Coordinates object that contains the current robot coordinates using the `get_current_coordinates` function.

```
current_coordinates = get_current_coordinates()
```

property position

The position part of the Coordinates object.

Returns:

The position in the form [x, y, z] in [mm].

Return type:

list of float

property orientation

The orientation part of the Coordinates object.

Returns:

The orientation in the form [roll, pitch, yaw] in [deg].

Return type:

list of float

to_list()

Convert the coordinates object to a list of coordinate values in the form [x, y, z, roll, pitch, yaw], where positional values are in [mm] and rotational values are in [deg].

Returns:

Coordinate values in list format.

Return type:

list of float

Examples:

This function can be used when the Coordinates are needed as a list:

```
coord = Coordinates({"x": 700.0, "y": -125.0, "z": 500.0,  
                    "roll": 180.0, "pitch": 0.0, "yaw": -90.0})  
  
# convert to list  
coord_list = coord.to_list()
```

to_dict()

Convert the coordinates object to a dict of coordinate values in the form {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}, where positional values are in [mm] and rotational values are in [deg].

Returns:

Coordinate values in dict format.

Return type:

dict of float

Examples:

This function can be used when the Coordinates are needed as a dict:

```
coord = Coordinates([700.0, -125.0, 500.0, 180.0, 0.0, -90.0])

# convert to dict
coord_dict = coord.to_dict()
```

inverted()

Calculates the inverse transform to this coordinates object.

Returns:

inverse of transform object

Return type:

Transform

Examples:

```
c = Coordinates([100, 200, 300, 15, -15, 30])

# create the inverse transform
c_inv = c.inverted()

# the inverse transform applied to the original transform
# this results in the identity transform
assert c * c_inv == Coordinates([0, 0, 0, 0, 0, 0])
assert c_inv * c == Coordinates([0, 0, 0, 0, 0, 0])
```

class Pose

The Pose object defines a pose of the robot, consisting of its coordinates (position & orientation) and its configuration (since the same set of coordinates can be reached with multiple configurations).

`__init__(pose)`

Create a Pose object, either by loading a saved pose by name or ID, or from the current pose of the robot.

Parameters:

pose (*str*, *int*, or *Pose*) –

Reference to the pose to load, which can be one of the following:

- the name (*str*) of a pose saved in the database
- the ID (*int*) of a pose saved in the database
- a Pose object (copying another Pose object)

Constructor:

Load a pose from the database.

```
pose = Pose("start_pose")
```

Create a Pose object that contains the current robot pose using the `get_current_pose` function.

```
current_pose = get_current_pose()
```

Create a Pose object that contains the specified coordinates and configuration using the `create_pose` function.

```
pose = create_pose(coordinates, configuration)
```

property coordinates

The coordinates of this pose, which is a Coordinates object containing the position and orientation values.

Returns:

The coordinates of this pose.

Return type:

[Coordinates](#)

property configuration

Since one set of pose coordinates can be reached with up to 8 possible configurations of the robot joints, the configuration is defined as a number (from 0 to 7) or a string (from 'c1' to 'c8') to uniquely define the pose.

Returns:

The index of the pose configuration (from 0 to 7).

Return type:

int

***class* LinearSegment**

The LinearSegment object describes a linear path segment of the TCP (tool center point) to the desired coordinates. Additionally, it includes information about the desired linear velocity, linear acceleration and the blend radius at the start of the segment. This blend radius allows for a deviation from the linear path to ensure a smooth transition between segments.

***__init__*(linear_segment)**

Create a LinearSegment object.

Parameters:

- **coordinates** (*list, dict, or [Coordinates](#)*) – Target coordinates.
- **linear_velocity** (*float or None*) – Desired linear velocity [mm/s]. If no linear velocity is defined, the global linear velocity value will be used.
- **linear_acceleration** (*float or None*) – Desired linear acceleration [mm/s²]. If no linear acceleration is defined, the global linear acceleration value will be used.
- **blend_radius** (*float or None*) – Blend radius at the start of the segment [mm]. If no blend radius is defined, 0.0 will be used instead.

Constructor:

Creating a LinearSegment object by defining all parameters:

```
linear_seg = LinearSegment([100, 200, 300, 15, -15, 30],  
                           linear_velocity=500, linear_acceleration=1000,  
                           blend_radius=100)
```

or using the default values:

```
linear_seg = LinearSegment([100, 200, 300, 15, -15, 30])
```

property coordinates

The target coordinates of this segment, which is a Coordinates object containing the position and orientation values.

Returns:

The target coordinates of this segment.

Return type:

[Coordinates](#)

property blend_radius

The blend radius at the start of this segment.

Returns:

The start blend radius.

Return type:

float

property linear_acceleration

The desired linear acceleration of this segment. If it is None, the global linear acceleration value is used instead.

Returns:

The desired linear acceleration.

Return type:

float or None

property linear_velocity

The desired linear velocity of this segment. If it is None, the global linear velocity value is used instead.

Returns:

The desired linear velocity.

Return type:

float or None

***class* CircularSegment**

The CircularSegment object describes a circular motion of the TCP (tool center point) through the intermediate position to the desired coordinates. Note that the orientation of that intermediate pose is defined by the algorithm. Additionally, it includes information about the desired linear velocity, linear acceleration and the blend radius at the start of the segment. This blend radius allows for a deviation from the circular path to ensure a smooth transition between segments.

`__init__(circular_segment)`

Create a CircularSegment object.

Parameters:

- **intermediate_position** (*list*) – An intermediate position [x, y, z] on the path towards the target coordinates. The intermediate position determines the curvature of the circular motion.
- **coordinates** (*list, dict or [Coordinates](#)*) – Target coordinates.
- **linear_velocity** (*float or None*) – Desired linear velocity [mm/s]. If no linear velocity is defined, the global linear velocity value will be used.
- **linear_acceleration** (*float or None*) – Desired linear acceleration [mm/s²]. If no linear acceleration is defined, the global linear acceleration value will be used.
- **blend_radius** (*float or None*) – Blend radius at the start of the segment [mm]. If no blend radius is defined, 0.0 will be used instead.

Constructor:

Creating a CircularSegment object by defining all parameters:

```
circular_seg = CircularSegment([0, 300, 300], [100, 200, 300, 15, -15, 30],  
                                linear_velocity=500, linear_acceleration=1000,  
                                blend_radius=100)
```

or using the default values:

```
circular_seg = CircularSegment([0, 300, 300], [100, 200, 300, 15, -15, 30])
```

***property* intermediate_position**

The intermediate position of this segment. It includes the x, y, z coordinates: [x, y, z]

Returns:

The intermediate position of this segment.

Return type:

list

***property* coordinates**

The target coordinates of this segment, which is a Coordinates object containing the position and orientation values.

Returns:

The target coordinates of this segment.

Return type:

[Coordinates](#)

***property* blend_radius**

The blend radius at the start of this segment.

Returns:

The start blend radius.

Return type:

float

***property* linear_acceleration**

The desired linear acceleration of this segment. If it is None, the global linear acceleration value is used instead.

Returns:

The desired linear acceleration.

Return type:

float or None

***property* linear_velocity**

The desired linear velocity of this segment. If it is None, the global linear velocity value is used instead.

Returns:

The desired linear velocity.

Return type:

float or None

***class* OrientationSegment**

The OrientationSegment object describes a pure orientation path segment where the TCP (tool center point) rotates according to the desired tool orientation while keeping its position constant. Additionally, it includes information about the desired angular velocity, angular acceleration and the blend radius at the start of the segment. If the blend radius is larger than 0, the rotation of the TCP start before reached the target position.

`__init__(orientation_segment)`

Create an OrientationSegment object.

Parameters:

- **orientation** (*list*) – Target orientation. Note that the position stays constant.
- **angular_velocity** (*float or None*) – Desired angular velocity [deg/s]. If no angular velocity is defined, the global angular velocity value will be used.
- **angular_acceleration** (*float or None*) – Desired angular acceleration [deg/s²]. If no angular acceleration is defined, the global angular acceleration value will be used.
- **blend_radius** (*float or None*) – Blend radius at the start of the segment [mm]. A blend radius of more than 0 allows the tool center point (TCP) to rotate before the position of this segment is reached. If no blend radius is defined, 0.0 will be used instead.

Constructor:

Creating an OrientationSegment object by defining all parameters:

```
orientation_seg = OrientationSegment([15, -15, 30],  
                                     angular_velocity=250,  
                                     angular_acceleration=500,  
                                     blend_radius=100)
```

or using the default values:

```
orientation_seg = OrientationSegment([15, -15, 30])
```

property orientation

The target orientation of this segment. Given as nautical angles [roll, pitch, yaw] in degrees

Returns:

The target orientation of this segment.

Return type:

list

property angular_velocity

The desired angular velocity of this segment. If it is None, the global angular velocity value is used instead.

Returns:

The desired angular velocity.

Return type:

float or None

property angular_acceleration

The desired angular acceleration of this segment. If it is None, the global angular acceleration value is used instead.

Returns:

The desired angular acceleration.

Return type:

float or None

property blend_radius

The blend radius at the start of this segment.

Returns:

The start blend radius.

Return type:

float

***class* CoordinatesSegment**

The CoordinatesSegment object describes a linear motion in joint space to the target coordinates. The TCP (tool center point) in contrast does generally not move linearly. Additionally, it includes information about the desired linear velocity, linear acceleration and the blend radius at the start of the segment. This blend radius allows for a deviation from the path to ensure a smooth transition between segments.

`__init__(coordinates_segment)`

Create a CoordinatesSegment object.

Parameters:

- **coordinates** (*list, dict or [Coordinates](#)*) – Target coordinates.
- **linear_velocity** (*float or None*) – Desired linear velocity [mm/s]. If no linear velocity is defined, the global linear velocity value will be used.
- **linear_acceleration** (*float or None*) – Desired linear acceleration [mm/s²]. If no linear acceleration is defined, the global linear acceleration value will be used.
- **blend_radius** (*float or None*) – Blend radius at the start of the segment [mm]. If no blend radius is defined, 0.0 will be used instead.

Constructor:

Creating a CoordinatesSegment object by defining all parameters:

```
coordinates_seg = CoordinatesSegment([100, 200, 300, 15, -15, 30],  
                                     linear_velocity=250,  
                                     linear_acceleration=500,  
                                     blend_radius=100)
```

or using the default values:

```
coordinates_seg = CoordinatesSegment([100, 200, 300, 15, -15, 30])
```

property coordinates

The target coordinates of this segment, which is a Coordinates object containing the position and orientation values.

Returns:

The target coordinates of this segment.

Return type:

[Coordinates](#)

property blend_radius

The blend radius at the start of this segment.

Returns:

The start blend radius.

Return type:

float

property linear_acceleration

The desired linear acceleration of this segment. If it is None, the global linear acceleration value is used instead.

Returns:

The desired linear acceleration.

Return type:

float or None

property linear_velocity

The desired linear velocity of this segment. If it is None, the global linear velocity value is used instead.

Returns:

The desired linear velocity.

Return type:

float or None

***class* PoseSegment**

The Pose object describes a linear motion in joint space to the target pose. The TCP (tool center point) in contrast does generally not move linearly. Additionally, it includes information about the desired linear velocity, linear acceleration and the blend radius at the start of the segment. This blend radius allows for a deviation from the path to ensure a smooth transition between segments.

`__init__(pose_segment)`

Create a PoseSegment object.

Parameters:

- **pose** (*Pose*) – Target pose.
- **linear_velocity** (*float or None*) – Desired linear velocity [mm/s]. If no linear velocity is defined, the global linear velocity value will be used.
- **linear_acceleration** (*float or None*) – Desired linear acceleration [mm/s²]. If no linear acceleration is defined, the global linear acceleration value will be used.
- **blend_radius** (*float or None*) – Blend radius at the start of the segment [mm]. If no blend radius is defined, 0.0 will be used instead.

Constructor:

Creating a PoseSegment object by defining all parameters:

```
pose_seg = PoseSegment(create_pose([100, 200, 300, 15, -15, 30], "c1"),
                        linear_velocity=250, linear_acceleration=500,
                        blend_radius=100)
```

or using the default values and pose from database:

```
pose_seg = PoseSegment("target_pose")
```

property pose

The target pose of this segment, which is a Pose object containing the position, orientation and configuration values.

Returns:

The target pose of this segment.

Return type:

Pose

property blend_radius

The blend radius at the start of this segment.

Returns:

The start blend radius.

Return type:

float

property linear_acceleration

The desired linear acceleration of this segment. If it is None, the global linear acceleration value is used instead.

Returns:

The desired linear acceleration.

Return type:

float or None

property linear_velocity

The desired linear velocity of this segment. If it is None, the global linear velocity value is used instead.

Returns:

The desired linear velocity.

Return type:

float or None

class Path

The Path object defines how the robot moves between two or more poses. A continuous path is defined only in tool space by multiple segments such as linear or circular. A path is called seamless, if the discrete solutions to the path do not require the robot to reposition itself (configuration change). A path is called continuous, if the acceleration is continuous along the path. A path starts at a specified pose (as opposed to coordinates). Segments of the path usually define coordinates, as the configuration of the path is defined by the initial pose.

`__init__(path)`

Create a Path object, either by loading a saved path by name or ID.

Parameters:

path (*str, int, or Path*) –

Reference to the path to load, which can be one of the following:

- the name (*str*) of a path saved in the database
- the ID (*int*) of a path saved in the database
- a Path object (copying another Path object)

Constructor:

Load a path from the database.

```
path = Path("pick_place_path")
```

Create a Path object that contains the specified initial pose and the defined path segments using the `create_path` function. Note that segments can also be added later on.

```
path = create_path(initial_pose, segments)
```

property initial_pose

The initial robot pose of the path.

Returns:

Initial robot pose.

Return type:

[Pose](#)

property segments

The segments included in the path. Note that the segments do not all have to be of the same type.

Returns:

Included segments.

Return type:

list of Segments ([LinearSegment](#), [CircularSegment](#), [OrientationSegment](#), [CoordinatesSegment](#), [PoseSegment](#))

add_segment(segment)

Adds a segment to the end of the path.

Parameters:

segment ([LinearSegment](#), [CircularSegment](#), [OrientationSegment](#), [CoordinatesSegment](#), [PoseSegment](#)) – Segment to be added.

Examples:

A segment can be added as follows:

```
# creating and adding a segment
lin_seg = LinearSegment([700.0, -125.0, 500.0, 180.0, 0.0, -90.0])
path.add_segment(lin_seg)
```

```
add_linear(coordinates, linear_velocity=None, linear_acceleration=None,
blend_radius=None)
```

Adds a linear segment to the end of the path.

Parameters:

- **coordinates** (*list, dict or [Coordinates](#)*) – Target coordinates of the path.
- **linear_velocity** (*float*) – Maximum allowed linear tool velocity [mm/s].
- **linear_acceleration** (*float*) – Maximum allowed linear tool acceleration [mm/s²].
- **blend_radius** (*float*) – Allowed blending radius [mm] at the start of the segment.

Examples:

A linear segment can be added as follows:

```
# add a linear segment with different argument types for the "coordinates"
parameter
path.add_linear([700.0, -125.0, 500.0, 180.0, 0.0, -90.0])
path.add_linear({"x": 700.0, "y": -125.0, "z": 500.0,
                 "roll": 180.0, "pitch": 0.0, "yaw": -90.0})
path.add_linear(Coordinates([700.0, -125.0, 500.0, 180.0, 0.0, -90.0]))
```

In these examples, no optional parameters were specified. Therefore, the global parameters are used. The parameters can also be explicitly defined in the function call:

```
# first linear segment is added
coord = Coordinates([700.0, -250.0, 500.0, 180.0, 0.0, -90.0])
path.add_linear(coord, linear_velocity=500.0, linear_acceleration=1000.0)

# second linear segment is added
path.add_linear([700.0, 250.0, 500.0, 180.0, 0.0, -90.0],
                 linear_velocity=250.0, linear_acceleration=500.0,
                 blend_radius=100.0)
```

The first segment defines a linear movement towards the specified target coordinates with a maximum velocity of 500 mm/s and a maximum acceleration of 1000 mm/s². Note that these coordinates are not reached exactly as the consecutive segment defines a blend radius of 100 mm, which allows a deviation from those coordinates to ensure a smoother and more efficient transition. The second segment then adds a linear movement to the next coordinates with a maximum velocity of 250 mm/s and a maximum acceleration of 500 mm/s².

```
add_circular(intermediate_position, coordinates, linear_velocity=None,  
linear_acceleration=None, blend_radius=None)
```

Adds a circular segment to the end of the path.

Parameters:

- **intermediate_position** (*list*) – An intermediate position ([x, y, z]) on the path towards the target coordinates. The intermediate position determines the curvature of the circular motion.
- **coordinates** (*list, dict or [Coordinates](#)*) – Target coordinates of the path.
- **linear_velocity** (*float*) – Maximum allowed linear tool velocity [mm/s].
- **linear_acceleration** (*float*) – Maximum allowed linear tool acceleration [mm/s²].
- **blend_radius** (*float*) – Allowed blending radius [mm] at the start of the segment.

Examples:

A circular segment can be added as follows:

```
# add a circular segment with different argument types for the "coordinates"  
parameter  
path.add_circular([0.0, -500.0, 500.0], [-500.0, -125.0, 500.0, 180.0, 0.0,  
-90.0])  
path.add_circular([0.0, -500.0, 500.0], {"x": -500.0, "y": -125.0, "z": 500.0,  
"roll": 180.0, "pitch": 0.0, "yaw": -90.0})  
path.add_circular([0.0, -500.0, 500.0],  
Coordinates([-500.0, -125.0, 500.0, 180.0, 0.0, -90.0]))
```

In these examples, no optional parameters were specified. Therefore, the global parameters are used. The parameters can also be explicitly defined in the function call:

```
# first circular segment is added  
coord = Coordinates([-500.0, 250.0, 500.0, 180.0, 0.0, -90.0])  
path.add_circular([-700.0, 0.0, 500.0], coord,  
linear_velocity=500.0, linear_acceleration=1000.0)  
  
# second circular segment is added  
path.add_circular([-700.0, 0.0, 500.0], [-500.0, -250.0, 500.0, 180.0, 0.0,  
-90.0],  
linear_velocity=250.0, linear_acceleration=500.0,  
blend_radius=0.0)
```

The first segment defines a circular movement through the specified intermediate position towards the target coordinates with a maximum velocity of 500 mm/s and a maximum acceleration of 1000 mm/s². Note that these coordinates are not reached exactly as the consecutive segment defines a blend radius of 100 mm, which allows a deviation from those coordinates to ensure a smoother and more efficient transition. The second segment then

adds a circular movement through the next intermediate position towards the next target coordinates with a maximum velocity of 250 mm/s and a maximum acceleration of 500 mm/s².

```
add_orientation(coordinates, angular_velocity=None, angular_acceleration=None,
blend_radius=None)
```

Adds an orientation segment to the end of the path.

Parameters:

- **coordinates** (*list, dict or [Coordinates](#)*) – Target coordinates of the path.
- **angular_velocity** (*float*) – Maximum allowed angular tool speed [deg/s].
- **angular_acceleration** (*float*) – Maximum allowed angular tool acceleration [deg/s²].
- **blend_radius** (*float*) – Allowed blending radius [mm] at the start of the segment. A value larger than 0 means it can already rotate before reaching the desired position of this segment.

Examples:

An orientation segment can be added as follows:

```
# add an orientation segment
path.add_orientation([150.0, 0.0, -90.0])
```

In these examples, no optional parameters were specified. Therefore, the global parameters are used. The parameters can also be explicitly defined in the function call:

```
# first orientation segment is added
path.add_orientation([180.0, 0.0, -90.0],
                    angular_velocity=250.0, angular_acceleration=500.0)

# second orientation segment is added
path.add_orientation([150.0, 0.0, -90.0], angular_velocity=500.0,
                    angular_acceleration=1000.0, blend_radius=100.0)
```

The first segment defines a rotation to the specified target orientation with a maximum velocity of 250 deg/s and a maximum acceleration of 500 deg/s². Note that, when running this path, this first rotation will be skipped as the blend radius of the consecutive segment is greater than 0 (see `OrientationSegment.__init__`) and it will directly execute the second segment. This segment defines a rotation to the target orientation of [150.0, 0.0, -90.0] with a maximum velocity of 500 deg/s and a maximum acceleration of 1000 deg/s². In contrast, if the blend radius of the second segment were set to 0.0, both rotations would be executed in sequence.

```
add_coordinates(coordinates, linear_velocity=None, linear_acceleration=None,
blend_radius=None)
```

Adds a coordinates segment to the end of the path.

Parameters:

- **coordinates** (*list, dict or [Coordinates](#)*) – Target coordinates of the path.
- **linear_velocity** (*float*) – Maximum allowed linear tool velocity [mm/s].
- **linear_acceleration** (*float*) – Maximum allowed linear tool acceleration [mm/s²].
- **blend_radius** (*float*) – Allowed blending radius [mm] at the start of the segment.

Examples:

A coordinates segment can be added as follows:

```
# different argument types for the "coordinates" parameter
path.add_coordinates([700.0, -125.0, 500.0, 180.0, 0.0, -90.0])
path.add_coordinates({"x": 700.0, "y": -125.0, "z": 500.0,
                    "roll": 180.0, "pitch": 0.0, "yaw": -90.0})
path.add_coordinates(Coordinates([700.0, -125.0, 500.0, 180.0, 0.0, -90.0]))
```

In these examples, no optional parameters were specified. Therefore, the global parameters are used. The parameters can also be explicitly defined in the function call:

```
# first coordinates segment is added
coord = Coordinates([700.0, -250.0, 500.0, 180.0, 0.0, -90.0])
path.add_coordinates(coord, linear_velocity=500.0, linear_acceleration=1000.0)

# second coordinates segment is added
path.add_coordinates([700.0, 250.0, 500.0, 180.0, 0.0, -90.0],
                    linear_velocity=250.0,
                    linear_acceleration=500.0, blend_radius=100.0)
```

The first segment defines a movement towards the specified target coordinates with a maximum velocity of 500 mm/s and a maximum acceleration of 1000 mm/s². Note that these coordinates are not reached exactly as the consecutive segment defines a blend radius of 100 mm, which allows a deviation from those coordinates to ensure a smoother and more efficient transition. The second segment then adds a movement to the next coordinates with a maximum velocity of 250 mm/s and a maximum acceleration of 500 mm/s².

`add_pose(pose, linear_velocity=None, linear_acceleration=None, blend_radius=None)`

Adds a pose segment to the end of the path.

Parameters:

- **pose** (*Pose*) – Target coordinates of the path.
- **linear_velocity** (*float*) – Maximum allowed linear tool velocity [mm/s].
- **linear_acceleration** (*float*) – Maximum allowed linear tool acceleration [mm/s²].
- **blend_radius** (*float*) – Allowed blending radius [mm] at the start of the segment.

Examples:

A pose segment can be added as follows:

```
# using a pose from the database
pose_db = get_pose("pick_pose")
path.add_pose(pose_db)

# using a newly created pose
new_pose = create_pose([700.0, -125.0, 500.0, 180.0, 0.0, -90.0], "c8")
path.add_pose(new_pose)
```

In these examples, no optional parameters were specified. Therefore, the global parameters are used. The parameters can also be explicitly defined in the function call:

```
# first pose segment is added
new_pose = create_pose([700.0, -125.0, 500.0, 180.0, 0.0, -90.0], "c8")
path.add_pose(new_pose, linear_velocity=500.0, linear_acceleration=1000.0)

# second pose segment is added
path.add_pose(get_pose("pick_pose"), linear_velocity=250.0,
              linear_acceleration=500.0, blend_radius=100.0)
```

The first segment defines a movement towards the specified target pose with a maximum velocity of 500 mm/s and a maximum acceleration of 1000 mm/s². Note that this pose is not reached exactly as the consecutive segment defines a blend radius of 100 mm, which allows a deviation from the corresponding coordinates to ensure a smoother and more efficient transition. The second segment then adds a movement to the next pose with a maximum velocity of 250 mm/s and a maximum acceleration of 500 mm/s².

***class* Socket**

The Socket is an object that is returned by the `create_tcp_client`, `create_tcp_server`, and `create_udp_socket` functions. Use this object to send and receive data via the socket.

`receive(timeout=None)`

Receiving a message through this socket.

Parameters:

timeout (*None or float*) – The maximum time to wait for a message. If not specified, wait indefinitely.

Returns:

The received message, or an empty string if a timeout occurred.

Return type:

str

Examples:

Waiting for an answer for 3 seconds, and print it when it is being received.

```
print(socket.receive(timeout=3.0))
```

`send(message)`

Sending a message through this socket.

Parameters:

message (*str*) – The message to send.

Examples:

Send a message to all connected peers.

```
socket.send("Hello world!")
```

close()

Closing the socket connection.

Examples:

Close the socket.

```
socket.close()
```

Add-on Functions

Cognex Add-on Functions

class Cognex

The Cognex module supports the usage of Cognex cameras. Note that configuring the Cognex camera requires the installation of the [Cognex In-Sight Explorer](#).

For more information about how to properly set up and connect your Cognex camera with the robot, please refer to the user manual.

All functions of this add-on are called using the `cognex.` prefix.

`connect_to_camera(ip, user='admin', password='')`

Setting up a TCP client connecting to the connected Cognex camera over the ISE Port for native commands.

Parameters:

- **ip** (*str*) – The IP address of the camera, e.g. `'10.0.0.210'`.
- **user** (*str*) –
The username to log into the camera (as previously set in [Cognex In-Sight Explorer](#)). The default username is `'admin'`.
- **password** (*str*) –
The password to log into the camera (as previously set in [Cognex In-Sight Explorer](#)). The default password is an empty string.

Raises:

MinorError – if the credentials are wrong, the connection is aborted during connection, or the connection took too long.

Examples:

Connect to a freshly delivered camera with the default IP and credentials.

```
cognex.connect_to_camera("10.0.0.210")
```

Connect to a camera with previously modified IP address and credentials.

```
cognex.connect_to_camera("192.168.80.235", user="Developer",  
password="my_password_1234")
```

trigger_camera(*ip*)

Setting the camera trigger event ("SW8") which takes a new picture and returning the data processed by the active Cognex job.

Parameters:

ip (*str*) – The IP address of the camera, e.g. `'10.0.0.210'`.

Returns:

The response message from the camera, containing detected entities.

Return type:

`str`

change_job(ip, job_name)

Changing the Cognex job via the native camera command "LF", which loads a job file already stored on the camera. Note that this command can take several seconds to execute.

Parameters:

- **ip** (*str*) – The IP address of the camera, e.g. `'10.0.0.210'`.
- **job_name** (*str*) –
The name of the job to switch to (as previously set in [Cognex In-Sight Explorer](#)).

Raises:

MinorError – if the camera does not exist, or changing the job failed.

Examples:

Change to a job previously stored as 'CircleDetection.job', and then take a picture to detect circles in the current camera view.

```
ip_address = "10.0.0.210"  
cognex.change_job(ip, "CircleDetection")  
detected_circle_message = cognex.trigger_camera(ip)
```

get_camera_message(ip, event_id, keyword, timeout=None)

Trigger an event on the camera, and requesting a specific keyword socket message on Cognex side.

Parameters:

- **ip** (*str*) – The IP address of the camera, e.g. `'10.0.0.210'`.
- **event_id** (*int*) – The ID of the Cognex event to trigger.
- **keyword** (*str*) – The keyword to wait for.
- **timeout** (*float or None*) – The timeout of the wait request (in [s]). Use `None` to wait indefinitely.

Raises:

TimeoutError – if the camera did not respond a valid string before the timeout expired.

Examples:

Requests the message containing the string 'my_message', linked to the event 1 (SW1) from the Cognex In-Sight job.

```
message = cognex.get_camera_message("10.0.0.210", 1, "my_message")
```

disconnect_camera(*ip*)

Disconnecting a specific camera.

Parameters:

ip (*str*) – The IP address of the camera, e.g. `'10.0.0.210'` .

Modbus Add-on Functions

class Modbus

The Modbus module contains functions for sending requests to an external REST server.

All functions of this add-on are called using the `modbus.` prefix.

write_user_register(*address, values*)

Writing to the output Modbus registers.

Parameters:

- **address** (*int*) – The address of the register to start writing. Possible values range from 0x10 (16) to 0xFF (255).
- **values** (*int or list of int*) – The value or values to be written. Each register can store up to 2 bytes of data. The maximum value to be stored is therefore 65535 ($=2^{16}-1$). If multiple values are provided, these values are written into consecutive registers.

Examples:

Write a single value to a specific address of the Modbus output registers.

```
# write 11111 to the address 0x20
modbus.write_user_register(0x20, 11111)
```

Write multiple values to consecutive output registers using a single command.

```
# write 555 to the address 0x30, and 777 to the address 0x31
modbus.write_user_register(0x30, [555, 777])
```

read_user_register(*address*, *count*=1)

Reading from the input Modbus registers.

Parameters:

- **address** (*int*) – The address of the register to start reading. Possible values range from 0x10 (16) to 0xFF (255).
- **count** (*int*) – The number of consecutive registers to be read. By default, read only one register.

Returns:

The values read from the registers. The length of this list is equal to the *count* argument.

Return type:

list of int

Examples:

Read a single value from a specific address of the Modbus input registers.

```
# read value from the address 0x20
value = modbus.read_user_register(0x20)[0]
```

Read multiple values from consecutive input registers using a single command.

```
# read values from the addresses 0x30 and 0x31
values = modbus.read_user_register(0x30, 2)
```

wait_for_event(*index*)

Waiting until the event with the specified index is set. Events are set from the outside by the Modbus client.

Parameters:

index (*int*) – The index of the event to wait for. Possible indices are from 0 to 15.

Examples:

Wait for an event to be set.

```
modbus.wait_for_event(2)
```

REST Add-on Functions

class Rest

The REST module contains functions for sending requests to an external REST server.

All functions of this add-on are called using the `rest.` prefix.

setup_client_connection(*target_url*)

Setting up a client connection for sending requests to a REST server.

Parameters:

target_url (*str*) – The URL (general API endpoint) of the REST server to connect to.

Returns:

The REST client connection object.

Return type:

[RestClientConnection](#)

Examples:

Establish a connection to a REST server.

```
connection = rest.setup_client_connection("https://some_url:5000/endpoint")
# ... this is similar to ...
connection = rest.setup_client_connection("https://some_url:5000/endpoint/")
```

***class* RestClientConnection**

The RestClientConnection is an object returned by `rest.setup_client_connection`. It is used to send requests to the connected REST server.

get_default_headers(*method*)

Reading the default headers that are sent with every request.

Parameters:

method (*str*) – The method ("get" or "post") for which to get the headers.

Returns:

The currently set default headers for this request method.

Return type:

dict

Examples:

Reading the default headers of the GET and POST methods.

```
connection = rest.setup_client_connection("https://some_url:5000/endpoint")
# reading the default headers of the GET method, which by default are set to
#    {'Accept': 'application/json'}
headers = connection.get_default_headers("get")
# reading the default headers of the POST method, which by default are set to
#    {'Accept': 'application/json', 'Content-Type': 'application/json'}
headers = connection.get_default_headers("post")
```

set_default_headers(*method*, *new_headers*)

Change the default headers that are sent with every request.

Parameters:

- **method** (*str*) – The method ("get" or "post") for which to set the headers.
- **new_headers** (*dict*) – The new default headers to overwrite.

Examples:

Establish a connection to a REST server.

```
connection = rest.setup_client_connection("https://some_url:5000/endpoint")
# resetting the headers of the GET method
connection.set_default_headers("get", {"Accept": "application/json"})
# resetting the headers of the POST method
connection.set_default_headers("post", {"Accept": "application/json",
"Content-Type": "application/json"})
```

```
get(resource, headers_override=None, query_params=None, secure_connection=True)
```

Sending a GET request to the REST server.

Parameters:

- **resource** – The name of the resource to access. The resource name is added to the URL (<url>/<resource>) and may optionally contain a leading '/' character.
- **headers_override** (*dict*) – Set the headers for this individual request. If not specified, it uses the default headers.
- **query_params** (*dict*) – Specifying a query (if any).
- **secure_connection** (*bool*) – Whether to use a secure connection (verify certificates) or not.

Returns:

The response code (HTTP code, e.g. `200` for successful request) and response payload (response data) (if any)

Return type:

(int, dict)

Examples:

Sending a GET request to a REST server, and checking its response code.

```
connection = rest.setup_client_connection("https://some_url:5000/endpoint")
code, payload = connection.get("my_resource") # or
connection.get("/my_resource")
# print received data if request was successful:
if code == 200:
    print(payload)
```

```
post(resource, data, headers_override=None, query_params=None,  
secure_connection=True)
```

Sending a POST request to the REST server.

Parameters:

- **resource** (*str*) – The name of the resource to access. The resource name is added to the URL (<url>/<resource>) and may optionally contain a leading '/' character.
- **data** (*str or dict*) – The body/data of the POST request.
- **headers_override** (*dict*) – Set the headers for this individual request. If not specified, it uses the default headers.
- **query_params** (*dict*) – Specifying a query (if any).
- **secure_connection** (*bool*) – Whether to use a secure connection (verify certificates) or not.

Returns:

The response code (HTTP code, e.g. `200` for successful request) and response payload (response data) (if any)

Return type:

(int, dict)

Examples:

Sending a POST request to a REST server, and checking its response code.

```
connection = rest.setup_client_connection("https://some_url:5000/endpoint")  
code, payload = connection.post("my_resource") # or  
connection.post("/my_resource")  
# print received data if request was successful:  
if code == 200:  
    print(payload)
```

ROS Add-on Functions

***class* ROSHandler**

The ROS module adds ROS functionalities for other add-ons and for direct access by users.

All functions of this add-on are called using the `ros_handler.` prefix.

ros_node()

The ROS module creates its own ROS node, which can be accessed directly through this function. This can be used to create objects like publishers, subscribers, etc. in scripts. Also, `rclpy` can be imported directly in applications for this purpose.

⚠ Warning

Do not run `rclpy.spin()` on this node, since this is already running in the background!

Examples:

Printing the names of all topics which the robot is publishing to.

```
for publisher in ros_handler.ros_node().publishers:
    print(publisher.topic_name)
```

Returns:

The ROS node of the ROS module.

Return type:

`rclpy.node.Node`